

MULTI: Debugging Command Reference



**Green Hills Software
30 West Sola Street
Santa Barbara, California 93101
USA
Tel: 805-965-6044
Fax: 805-965-6343
www.ghs.com**

LEGAL NOTICES AND DISCLAIMERS

GREEN HILLS SOFTWARE 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 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 to notify any person of such revision or changes.

Copyright © 1983-2016 by Green Hills Software. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Green Hills Software.

Green Hills, the Green Hills logo, CodeBalance, GMART, GSTART, INTEGRITY, MULTI, and Slingshot are registered trademarks of Green Hills Software. AdaMULTI, Built with INTEGRITY, EventAnalyzer, G-Cover, GHnet, GHnetLite, Green Hills Probe, Integrate, ISIM, u-velOSity, PathAnalyzer, Quick Start, ResourceAnalyzer, Safety Critical Products, SuperTrace Probe, TimeMachine, TotalDeveloper, DoubleCheck, and velOSity are trademarks of Green Hills Software.

All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

For a listing of Green Hills Software's periodically updated patent marking information, please visit http://www.ghs.com/copyright_patent.html.

PubID: debug_cmd-592071

Branch: <http://toolsvc/branches/release-branch-70>

Date: May 8, 2017

Contents

Preface	xv
About This Book	xvi
The MULTI 7 Document Set	xvii
Conventions Used in the MULTI Document Set	xviii
 1. Using Debugger Commands	 1
Availability of Debugger Commands	2
Getting Help Information about Debugger Commands	2
Finding Debugger Commands in This Book	3
Debugger Command Conventions	3
Using Address Expressions in Debugger Commands	5
Specifying Line Numbers	7
Name Resolution	8
Identifying Breakpoints in Debugger Commands	10
Breakpoint IDs and Labels	10
Breakpoint Ranges and Lists	11
Command Syntax	12
Using Command Lists in Debugger Commands	12
Continuing Commands onto Subsequent Lines	13
Including Comments in Debugger Commands	13
Terminating Commands	14
Default Search Path for Files Specified in Commands	14
 2. General Debugger Command Reference	 15
General Debugger Commands	16
asm	17
attach	18
caches	20
call	20

dbnew	20
debug	21
debugsnapshot	21
detach	23
eagleconfig	23
endtime	26
history	26
launcher	27
loadsym	27
mev	28
monitor	29
mrsv	30
new	31
output	32
P	33
q	35
quit	35
quitall	36
restore	37
save	37
starttime	38
unloadsym	38
wait	38

3. Breakpoint Command Reference 43

Breakpoint Commands	45
b	46
B	48
bA	49
bi, bI	49
bif	50
bpload	51
bpsave	51
bpview, breakpoints	52
bt	52
bu, bU	53
bx, bX	54
d	55

D	56
dz	56
edithwbp	58
editwbp	58
hardbrk	59
install_bp_on_request	64
rominithbp	65
sb	66
setbrk	67
sethbp	68
stopif	68
stopifi	69
tog	70
Tog	71
watchpoint	71

4. Building Command Reference 73

Building Commands	74
build	74
builder	74
wgutils	75

5. Call Stack Command Reference 77

Call Stack Commands	78
calls	78
callsvievw	79
cvconfig	81

6. Configuration Command Reference 83

General Configuration Commands	84
clearconfig	85
configoptions	85
configure	86
configurefile	86
fileextensions	87
fontsize	87
imagename	88

loadconfigfromfile	88
saveconfig	88
saveconfigtofile	89
setintegritydir	89
setuvelocitydir	90
source	90
sourceroot	91
star	92
syncolor	93
Button, Menu, and Mouse Commands	94
->	95
customizemenus	95
customizetoolbar	95
debugbutton, editbutton	96
inspect	98
keybind	98
menu	99
mouse	99

7. Debugger Note Command Reference 101

Debugger Note Commands	102
notedel	102
noteedit	103
notelist	103
notestate	104
noteview	104

8. Display and Print Command Reference 105

Display and Print Commands	106
asmmode	108
assem	108
cat	109
clear	109
comeback	109
components	110
dbprint	110
debugpane	111

dumpfile	111
E	112
echo	113
eval	113
examine	114
goaway	114
l	115
map	117
mprintf	117
mrulist	118
mute	119
p, print	119
printline	119
printphys	120
printwindow	120
pwd	120
Q	121
savedebugpane	121
windowcopy	122
windowpaste, windowspaste	122

9. Help and Information Command Reference 123

Help and Information Commands	124
about	124
aboutlic	125
bugreport	125
help	125
info	125
usage	126

10. Memory Command Reference 127

General Memory Commands	128
compare, compareb	129
copy, copyb	130
disassemble	131
fill, fillb	132
find, findb	133
flash	134

memdump	136
memload	137
memread	139
memtest	139
memwrite	142
verify	143

11. Navigation Command Reference 145

Navigation Commands	146
+	147
-	147
e	147
indexnext	148
indexprev	149
number	149
scrollcommand	149
switch	151
uptosource	151

12. Profiling Command Reference 153

profile	154
---------------	-----

13. Program Execution Command Reference 159

General Program Execution Commands	160
g	160
getargs	160
setargs	161
Continue Commands	162
bc	164
bcb	164
bcbhardbrk	165
bchardbrk	165
c	166
cb	166
cbhardbrk	167
cf	167
cfb	168

chardbrk	168
runbacktohardbrk	168
runtohardbrk	169
runtohere	169
Halt Commands	170
H	170
halt	170
k	171
Run Commands	171
r	171
R	172
rb, Rb	173
restart	173
resume	174
rundir	174
runtask	175
Signal Commands	176
signal	176
zignal	176
Single-Stepping Commands	177
bcU	179
bprev	180
bs	180
bsi	180
cu, cU	180
s	181
S, n	182
si	182
Si, ni	183
sl	183
Sl, nl	184
stepinto	184
Synchronous Debugging Commands	185
syncrc	185
Task Execution Commands	187
taskaction	187

14. Register Command Reference 189

Register Commands	190
regadd	191
regappend	191
regbasefile	192
regload	192
regtab	192
regunload	194
regvalload	194
regvalsave	194
regview	195

15. Scripting Command Reference 197

Command Manipulation and Debugger Macro Commands	198
alias	199
cedit	199
define	200
macrotrace	200
return	201
route	201
sc	201
shell	202
substitute	203
unalias	204
Conditional Program Execution Commands	204
break	205
continue	205
do	206
for	206
if	207
while	208
Dialog Commands	208
alertdialog	209
dialog	209
directorydialog	210
filedialog	210

External Tool Commands	211
evaltosocket	212
make	212
socket	212
History Commands	214
!	215
!!	215
backhistory	216
forwardhistory	216
h	217
Hook Commands	217
addhook	217
clearhooks	219
listhooks	220
MULTI-Python Script Commands	221
python, py	222
pywin	223
Object Structure Awareness (OSA) Commands	223
osacleanup	224
osacmd	225
osaexplorer	225
_osaFillGuiWithObj	226
osainject	227
osasetup	227
osatask	228
osatasklist	229
osaview	229
taskwindow	230
Record and Playback Commands	230
>	232
>>	232
<	233

16. Search Command Reference 235

Search Commands	236
/	237

?	237
bsearch	238
chgcase	238
completeselection	239
dialogsearch	239
fsearch	239
grep	240
isearch	241
isearchadd	242
isearchreturn	242
printsearch	243

17. Target Connection Command Reference 245

General Target Connection Commands	246
change_binding	247
connect	247
connectionview	250
disconnect	250
iobuffer	251
load	251
prepare_target	253
reset	256
set_runmode_partner	256
setup	257
target, xmit	258
targetinput, xmitio	259
unload	259
Serial Connection Commands	260
serialconnect	260
serialdisconnect	261

18. Task Group Command Reference 263

Task Group Commands	264
changegroup	264
creategroup	266
destroygroup	267
groupaction	267

listgroup	268
19. TimeMachine and INTEGRITY TimeMachine Command Reference	269
INTEGRITY TimeMachine Commands	270
tmhistory	270
TimeMachine and Trace Commands	271
timemachine	272
trace	273
tracebrowse	277
tracedata	277
tracefunction	278
traceline	278
traceload	278
tracemevsys	279
tracepath	280
tracepro	280
tracesave	280
tracesavetext	281
tracesubfunction	282
20. Tracepoint Command Reference	283
Tracepoint Commands	284
edittp	284
passive	285
tpdel	285
tpenable	286
tplist	286
tpprint	287
tppurge	287
tpreset	288
tpset	288
21. View Command Reference	291
General View Commands	292
browse	294
browseref, xref	295

diff	296
edit	296
editview	297
heapview	297
localsview	298
memview	299
showdef	299
showhistory	300
top	300
update	301
view	301
viewdel	303
viewlist	303
window	304
Cache View Commands	305
cachefind	305
cacheview	305
Data Visualization Commands	306
dataview	306
dvclear	306
dvload	307
dvprofile	307
A. Deprecated Command Reference	309
Deprecated Commands	310
Index	313

Preface

Contents

About This Book	xvi
The MULTI 7 Document Set	xvii
Conventions Used in the MULTI Document Set	xviii

This preface discusses the purpose of this manual, the MULTI documentation set, and typographical conventions used.

About This Book

This book provides information about using Debugger commands, and it provides a comprehensive listing of Debugger commands. It is divided into the following chapters:

- *Chapter 1: Using Debugger Commands* describes the conventions used to document the Debugger commands and explains some common concepts and procedures related to using Debugger commands. See Chapter 1, “Using Debugger Commands” on page 1.
- Remaining chapters document all commands that can be entered into the Debugger command pane. Most of these commands correspond to actions that can be performed from the graphical user interface, as described in Parts I through VI of the *MULTI: Debugging* book. The last chapter documents deprecated Debugger commands and the commands that have superseded them.



Note

New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your installation directory for release notes, **README** files, and other supplementary documentation.

The MULTI 7 Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Provides an introduction to the MULTI Integrated Development Environment (IDE) and leads you through a simple tutorial.
- *MULTI: Licensing* — Describes how to obtain, install, and administer MULTI licenses.
- *MULTI: Managing Projects and Configuring the IDE* — Describes how to create and manage projects and how to configure the MULTI IDE.
- *MULTI: Building Applications* — Describes how to use the compiler driver and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- *MULTI: Configuring Connections* — Describes how to configure connections to your target.
- *MULTI: Debugging* — Describes how to set up your target debugging interface for use with MULTI and how to use the MULTI Debugger and associated tools.
- *MULTI: Debugging Command Reference* — Explains how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- *MULTI: Scripting* — Describes how to create MULTI scripts. Also contains information about the MULTI-Python integration.

For a comprehensive list of the books provided with your MULTI installation, see the **Help** → **Manuals** menu accessible from most MULTI windows.

All books are available in one or more of the following formats:

- Print
- Online help, accessible from most MULTI windows via the **Help** → **Manuals** menu
- PDF, available in the **manuals** subdirectory of your IDE or Compiler installation

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI) modes.

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Indication	Example
bold type	Filename or pathname	C:\MyProjects
	Command	setup command
	Option	-G option
	Window title	The Breakpoints window
	Menu name or menu choice	The File menu
	Field name	Working Directory:
	Button name	The Browse button
italic type	Replaceable text	-o filename
	A new term	A task may be called a <i>process</i> or a <i>thread</i>
	A book title	<i>MULTI: Debugging</i>
monospace type	Text you should enter as presented	Type <code>help command_name</code>
	A word or words used in a command or example	The wait [-global] command blocks command processing, where -global blocks command processing for all MULTI processes.
	Source code	<code>int a = 3;</code>
	Input/output	<code>> print Test</code> <code>Test</code>
	A function	<code>GHS_System()</code>
ellipsis (...) (in command line instructions)	The preceding argument or option can be repeated zero or more times.	debugbutton [name]...

Convention	Indication	Example
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	> print Test Test
pipe () (in command line instructions)	One (and only one) of the parameters or options separated by the pipe or pipes should be specified.	call <i>func</i> <i>expr</i>
square brackets ([]) (in command line instructions)	Optional argument, command, option, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	.macro <i>name</i> [<i>list</i>]

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-*option*]...*filename*

The formatting indicates that:

- **gxyz** is a command.
- The option *-option* should either be replaced with one or more appropriate options or be omitted.
- The word *filename* should be replaced with the actual filename of an appropriate file.

The square brackets and the ellipsis should not appear in the actual command you enter.

Chapter 1

Using Debugger Commands

Contents

Availability of Debugger Commands	2
Getting Help Information about Debugger Commands	2
Finding Debugger Commands in This Book	3
Debugger Command Conventions	3
Using Address Expressions in Debugger Commands	5
Identifying Breakpoints in Debugger Commands	10
Command Syntax	12
Default Search Path for Files Specified in Commands	14

This chapter describes the conventions used to document the Debugger commands and explains some common concepts and procedures related to using Debugger commands. The remainder of this manual describes all of the MULTI Debugger commands. For a description of the command pane, the area of the main Debugger window in which you enter Debugger commands, and shortcuts that can be used in the command pane, see “The Command Pane” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

Availability of Debugger Commands

All of the Debugger commands can be executed from the Debugger command pane. Many of these commands can also be invoked using Debugger menus, buttons, hot keys, or mouse bindings, as noted in the description for each command.

Some Debugger commands are not available in non-GUI mode. These commands are labeled *GUI only*. If a command is not marked *GUI only*, it is available in both GUI and non-GUI mode. For more information about GUI and non-GUI modes, see “Starting the Debugger in GUI Mode” in Chapter 1, “Introduction” in the *MULTI: Debugging* book and “Starting the Debugger in Non-GUI Mode (Linux/Solaris only)” in Chapter 1, “Introduction” in the *MULTI: Debugging* book.

A few Debugger commands are available only on Linux/Solaris or while debugging Linux/Solaris-like targets, and are labeled *Linux/Solaris only*.

For a full explanation of other notational and usage conventions pertaining to the Debugger commands, see “Debugger Command Conventions” on page 3.

Getting Help Information about Debugger Commands

The MULTI Debugger provides two commands that display help about Debugger commands:

- `help command` — Opens the Help Viewer on documentation for the specified command.
- `usage command` — Prints the basic syntax of the specified command to the command pane.

For more information about these and other help commands, see Chapter 9, “Help and Information Command Reference” on page 123.

Finding Debugger Commands in This Book

The following chapters provide comprehensive documentation of all of the commands available for use in the command pane of the MULTI Debugger. The commands are grouped by function.

If you are looking for information about a specific command, consult the index for the specific location of the relevant documentation. For an alphabetical list of all Debugger commands, see the entry **commands** in the index.

For general information about using Debugger commands, and for an explanation of the conventions used in the command syntax and descriptions, see “Debugger Command Conventions” on page 3. For a description of the command pane, the area of the main Debugger window in which you enter Debugger commands, and shortcuts that can be used in the command pane, see “The Command Pane” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

Debugger Command Conventions

The following table describes symbols, placeholders, and concepts that are related to Debugger commands and that are used in the following chapters.

%bp_ID	Some Debugger commands accept one or more %bp_ID arguments, where <i>bp_ID</i> is the breakpoint identification number assigned to a breakpoint by MULTI when the breakpoint is created. For more information, see “Breakpoint IDs and Labels” on page 10. Breakpoint IDs in the form %bp_ID can be used to specify a single breakpoint, a breakpoint range, or a breakpoint list related to the execution of a specific command. For more information, see “Breakpoint Ranges and Lists” on page 11.
---------------	---

%bp_label	The b commands for setting breakpoints (for example, b, bi, bif, and bx) accept %bp_label as an argument, where <i>bp_label</i> is the user-specified name for the breakpoint. <i>bp_label</i> should not contain spaces or special characters. The following example sets a breakpoint labeled <code>foo</code> on line 24 of function <code>main</code>: <pre>> b %foo main#24</pre> After a breakpoint has been created with a breakpoint label, other commands (such as B, e, d, and tog) can use the breakpoint labels to refer to that specific breakpoint. For example: <pre>> d %foo</pre> removes the breakpoint labeled <code>foo</code>. Breakpoint labels can also be used to specify breakpoint lists and ranges. For more information, see “Breakpoint Ranges and Lists” on page 11.
@bp_count	The b commands for setting breakpoints (for example, b and bx) accept @bp_count as an argument, where <i>bp_count</i> is an integer. When a breakpoint's count is greater than 1, your process skips the breakpoint, and the count is decremented by 1. When the count reaches 1, the breakpoint stops your process in accordance with the breakpoint's other attributes. In this way, a breakpoint with a count can be used to stop your process less frequently and reproduce complex software conditions. Breakpoints with counts are especially useful for debugging inner loops.
@continue_count	Some continue commands use the @continue_count argument to specify how many breakpoints the Debugger will pass before stopping. For example, if continue_count is 4, the Debugger skips over the next 3 breakpoints and stops the process at the fourth breakpoint. Note: Only breakpoints that stop program execution are counted. A conditional breakpoint whose condition is false or a breakpoint whose commands resume a process are not counted. For more information, see “Continue Commands” on page 162.

{<i>commands</i>}	Curly braces indicate a command list, where <i>commands</i> can be either a single command or a list of commands with the syntax: <i>command; command; ...</i> Some Debugger commands, such as certain breakpoint commands, can be scripted. That is, they can accept command lists that specify other commands to be performed at specific times. For more information, see “Using Command Lists in Debugger Commands” on page 12.
<i>address_expression</i>	In the documentation describing the syntax for Debugger commands, <i>address_expression</i> is a placeholder for an expression referring to a specific location within your program. The Debugger accepts a variety of address expressions. For examples of common address expression formats, see “Using Address Expressions in Debugger Commands” on page 5.
GUI only	The label GUI only indicates that a command is not available in non-GUI mode. For more information about GUI and non-GUI modes, see “Starting the Debugger in GUI Mode” in Chapter 1, “Introduction” in the <i>MULTI: Debugging</i> book and “Starting the Debugger in Non-GUI Mode (Linux/Solaris only)” in Chapter 1, “Introduction” in the <i>MULTI: Debugging</i> book.
Linux/Solaris only	The label Linux/Solaris only indicates that the command is only available on Linux/Solaris or while debugging Linux/Solaris-like target systems.
<i>stacklevel_</i>	A number followed by an underscore refers to the specified level of the call stack. For example, if the function <code>main()</code> calls <code>foo()</code>, which calls <code>bar()</code>, which calls <code>hum()</code>, and in the Debugger you are currently debugging <code>hum()</code>, the following command: <pre>> e 2_</pre> changes the current viewing location to <code>foo()</code>, because <code>foo()</code> is at frame 2 in the current call stack.

Using Address Expressions in Debugger Commands

In the documentation describing the syntax for Debugger commands, *address_expression* is a placeholder for an expression referring to a specific location within your program. Note that address expressions differ from standard expressions, which are discussed in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

The following table provides example address expressions, which are specified here in conjunction with the **e** command. The **e** command navigates to the location indicated by the given address expression (see “e” on page 147).

**Note**

When the *address_expression* refers to a line number, the meaning of the expression varies depending on whether the configuration option **procRelativeLines** is on (function-relative mode) or off (file-relative mode). For more information about specifying line numbers, see “Specifying Line Numbers” on page 7.

Command	Effect
e 10	Function-relative mode: Examines line number 10 in the current function. File-relative mode: Examines line number 10 in the current file.
e +10 or -10	Examines ten lines after or before the current position, respectively.
e 0x1234	Examines the line at address 0x1234.
e func2	Examines function <code>func2</code> .
e func2#4	Function-relative mode: Examines line 4 of function <code>func2</code> . File-relative mode: Examines line 4 of the file containing <code>func2</code> , if that line exists within <code>func2</code> .
e "file3.c"#4	Examines line 4 of file file3.c .
e "file3.c"#func2	Examines function <code>func2</code> in file file3.c .
e "file3.c"#func2#4	Function-relative mode: Examines line 4 of function <code>func2</code> in file file3.c . File-relative mode: Examines line 4 of the file file3.c , if that line exists within <code>func2</code> .
e (foo)	Examines the address that is the value of the standard expression <code>foo</code> .
e global_variable	Examines the source location where <code>global_variable</code> is defined.

Command	Effect
<code>e (\$retadr())</code>	Examines the return address (exit point) of the current function. For a function without source line debug information, examines the address one byte past the end of the function's last instruction.
<code>e 1b</code>	Examines the location of the breakpoint with ID = 1.
<code>e %bp_label</code>	Examines the location of the breakpoint labeled <i>bp_label</i> .
<code>e 2_</code>	Examines stack level 2.
<code>e "file3.c"##func2##label4</code>	Examines C Label <i>label4</i> in function <i>func2</i> in file file3.c .
<code>e func2##label4</code>	Examines C Label <i>label4</i> in function <i>func2</i> .
<code>e ##label4</code>	Examines C Label <i>label4</i> in the current function.
<code>e *</code>	Examines the function list (wildcard search).
<code>e \$register_name</code>	Examines the address stored in the register <i>register_name</i> .
<code>e \$system_variable</code>	Examines the address stored in the system variable <i>system_variable</i> .

For information about the way MULTI resolves names in address expressions, see “Name Resolution” on page 8.

Specifying Line Numbers

The MULTI Debugger has two modes for interpreting line numbers: function-relative mode and file-relative mode. By default, the Debugger uses function-relative mode and interprets line numbers relative to a function rather than to a file. To change from one mode to another, toggle the option **Use function relative line numbers (vs. file relative)**, which appears on the **Debugger** tab of the **Options** window. (From the command pane, use the syntax `configure procRelativeLines on | off`.)

The following examples demonstrate how the same command will examine different lines, depending on whether the Debugger is configured for function-relative or file-relative mode.

Command	Effect in Function-Relative Mode	Effect in File-Relative Mode
<code>e func3#4</code>	Examines the source code at line 4 of function <code>func3</code> .	Examines the source code at line 4 of file containing function <code>func3</code> . (The line must exist within <code>func3</code> .)
<code>e 4</code>	Examines the source code at line number 4 in the current function.	Examines the source code at line number 4 in the current file.
<code>e #4</code>	Examines the source code at line number 4 in the current file.	Examines the source code at line number 4 in the current function.

Name Resolution

When resolving names in address expressions, MULTI attempts to match the name with a function, a global variable, or a program section. If attempting to match the name with a function, MULTI generally searches fully qualified function names first, then base names.

In C, the base name of a function and the fully qualified name are the same. In C++, the fully qualified name consists of the function's name and argument type(s), while the base name consists of the name only. Some examples follow.

```
int ex1(int arg1);
```

- Base name in C — `ex1`
- Fully qualified name in C — `ex1`
- Base name in C++ — `ex1`
- Fully qualified name in C++ — `ex1(int)`

```
int ex2();
```

- Base name in C — `ex2`
- Fully qualified name in C — `ex2`
- Base name in C++ — `ex2`
- Fully qualified name in C++ — `ex2()`

```
int ex3(int arg1, float arg2, short arg3, char * arg4);
```

- Base name in C — `ex3`

- Fully qualified name in C — `ex3`
- Base name in C++ — `ex3`
- Fully qualified name in C++ — `ex3(int, float, short, char *)`

Names in standard address expressions are resolved as follows.

If you specified a file in the address expression, MULTI attempts, in the order indicated below, to match the name with:

1. The fully qualified name of a function in the specified file
2. The base name of a function in the specified file
3. A global variable in the specified file

If you did not specify a file in the address expression, MULTI attempts, in the order indicated below, to match the name with:

1. The fully qualified name of a static function in the current file
2. The fully qualified name of a non-static function in any file
3. An unresolved link symbol (generally a function name in a library that is not yet loaded)
4. In C — The name of any function (including static functions) in the program

In C++ — The base name of any function in any class/namespace in the program (If the name in the expression is prefixed with `::`, MULTI searches only the global namespace for a match. If the name in the expression is fully qualified, MULTI uses the given qualifications in its search.)

If MULTI finds more than one matching result, you may be asked to choose one.

5. Any global variable (only if the name in the expression is *not* followed by `#`)
6. A section name on the target (only if the name in the expression is *not* followed by `#`)

Identifying Breakpoints in Debugger Commands

You can use breakpoint IDs, breakpoint labels, breakpoint ranges, and breakpoint lists to specify breakpoints in Debugger commands.

Breakpoint IDs and Labels

MULTI automatically assigns a *breakpoint ID* number to each breakpoint when it is created. Breakpoints are numbered sequentially in the order in which they were created, without respect to their location. To see a list of breakpoints by breakpoint ID, use the **B** command. (For a full description of the information printed about each breakpoint when this command is executed, see “B” on page 48.)

The breakpoint ID provides a way to refer to specific breakpoints. Some Debugger commands accept one or more **%bp_ID** arguments, where *bp_ID* is the breakpoint ID number. Breakpoint IDs in the form **%bp_ID** can also be used to specify a range of breakpoints or a list of breakpoints, as described in “Breakpoint Ranges and Lists” on page 11.

You can also specify a *breakpoint label* when you create a new breakpoint. Most of the **b** commands for setting breakpoints (for example, **b**, **bi**, **bif**, and **bx**) accept **%bp_label** as an argument to specify a name for a new breakpoint. *bp_label* should not contain spaces or special characters.

The following example sets a breakpoint labeled `foo` on line 24 of function `main`:

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

After a breakpoint has been created with a breakpoint label, other commands (such as **B**, **e**, **d**, and **tog**) can use the breakpoint label to refer to that specific breakpoint. For example:

```
> d %foo
```

removes the breakpoint labeled `foo`. For information about the **B**, **d**, and **tog** commands, see “Breakpoint Commands” on page 45. For information about the **e** command, see “Navigation Commands” on page 146.

To add a label after a breakpoint is created, use the **Software Breakpoint Editor**. For more information, see “Creating and Editing Software Breakpoints” in Chapter

8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

Like breakpoint IDs, breakpoint labels can also be used to specify breakpoint ranges and breakpoint lists, as described in the next section.

Breakpoint Ranges and Lists

A *breakpoint range* has the format:

`%bp_ID_or_bp_label_1:%bp_ID_or_bp_label_2`

and refers to all of the breakpoints with breakpoint IDs between the IDs for the breakpoints referred to by `bp_ID_or_bp_label_1` and `bp_ID_or_bp_label_2`, inclusive. For example, `%3:%6` refers to the four breakpoints with breakpoint IDs 3, 4, 5, and 6. You can specify breakpoint labels in a breakpoint range. These will be converted to the corresponding numerical ID in order to establish the range. The breakpoint ID of the second breakpoint in the range must be greater than or equal to the breakpoint ID of the first breakpoint in the range.

A *breakpoint list* is a comma-separated list of breakpoint IDs, breakpoint labels, and/or breakpoint ranges. You can include one or more breakpoint ranges in a breakpoint list. For example, the breakpoint list `%3,%6:%8` refers to the four breakpoints with identification numbers 3, 6, 7, and 8.

Some breakpoint commands, such as the **B** and **d** commands, can take breakpoint lists as arguments. For example:

Command	Effect
<code>d %foo,%bar,%gamma</code>	Removes the breakpoints labeled <code>foo</code> , <code>bar</code> , and <code>gamma</code> .
<code>d %foo:%gamma</code>	Removes the breakpoints with breakpoint IDs in the range between the breakpoint ID of breakpoint labeled <code>foo</code> and the breakpoint ID of the breakpoint labeled <code>gamma</code> . The breakpoint ID of the second breakpoint in the range must be greater than or equal to the ID of the first breakpoint in the range.
<code>d %1,%4:%7</code>	Removes the breakpoints with ID numbers 1, 4, 5, 6, and 7.

For more information about breakpoint IDs and breakpoint labels, see “Breakpoint IDs and Labels” on page 10. For information about the **B** and **d** commands, see “Breakpoint Commands” on page 45.

Command Syntax

Using Command Lists in Debugger Commands

Some Debugger commands, such as certain breakpoint commands and conditional execution commands, can be scripted. That is, they can contain a list of other commands to be performed when certain conditions are met. Such commands are specified in a *command list*, which is indicated by curly braces ({ }). A command list can contain either a single command or a list of commands with the syntax: { *command1*; *command2*; ... }.

A command list can span multiple lines. That is, after the initial curly brace ({), which must be included on the first line, you can continue to enter commands on separate lines until you end the list with a closing curly brace (}).

For example, a valid **if** command can be written to span multiple lines as follows:

```
if ($test == 0) {  
    mprintf("Invalid value\n");  
} else {  
    mprintf("Valid value\n");  
}
```

The following **if** command, which also spans multiple lines, uses invalid syntax (the curly braces begin new lines):

```
if ($test == 0)  
{  
    mprintf("Invalid value\n");  
} else  
{  
    mprintf("Valid value\n");  
}
```

Curly braces may contain other pairs of curly braces, as long as they are all paired and closed correctly. For example, the following command will set a scripted breakpoint that checks the value of some global variables.

```
MULTI> b {  
continued> mprintf("Fly = %d\n", fly);
```

```
continued> if (bar<9) {resume} else {echo Error: bar too high}  
continued> }
```

When the breakpoint is hit, the Debugger will first print:

```
Fly =
```

followed by the decimal value of the variable `fly`. Then, if the value of the variable `bar` is less than nine, the process will continue to run. Otherwise, the Debugger will print:

```
Error: bar too high
```

and the process will remain stopped at the breakpoint.

Continuing Commands onto Subsequent Lines

To issue a command that spans multiple lines:

- End every line but the last with a backslash (`\`), or
- Use an open curly brace (`{`) as documented in “Using Command Lists in Debugger Commands” on page 12.

Including Comments in Debugger Commands

You can use C or C++ style comments in Debugger commands. For example:

```
> echo foo /* This is a C style comment */  
foo  
> echo bar // This is a C++ style comment  
bar
```

These comments will be stripped out and ignored during command processing.



Note

C-style comments `/* */` are allowed to span lines.

Terminating Commands

All MULTI Debugger commands are assumed to be terminated by a new line or by a semicolon (;) unless the new line or semicolon occurs inside a command list (denoted by { }) or the new line is preceded by a backslash (\). See “Using Command Lists in Debugger Commands” on page 12.

Default Search Path for Files Specified in Commands

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

- Use the **source** command (see “source” on page 90).
- Use the **-I** command line option to MULTI. For more information, see “Using the Command Line” in Chapter 1, “Introduction” in the *MULTI: Debugging* book.
- 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.

Chapter 2

General Debugger Command Reference

Contents

General Debugger Commands	16
---------------------------------	----

The commands in this chapter allow you to perform common Debugger operations such as attaching to and detaching from a program, debugging a new program, blocking command processing, and closing the Debugger or the MULTI IDE. For information about using the Debugger, see the *MULTI: Debugging* book.

General Debugger Commands

The following list provides a brief description of each general Debugger command. For a command's arguments and for more information, see the page referenced.

- **asm** — Assembles the given assembly instruction into machine code and prints this machine code (see “asm” on page 17).
- **attach** — Attaches to a process running on an RTOS (see “attach” on page 18).
- **caches** — Enables or disables the use of the memory cache and the assembly cache (see “caches” on page 20).
- **call** — Prints the value of the given function or expression (see “call” on page 20).
- **dbnew** — Allows you to debug a different program (see “dbnew” on page 20).
- **debug** — Replaces the current program with the given program (see “debug” on page 21).
- **debugsnapshot** — Saves a debug snapshot to a file on the host (see “debugsnapshot” on page 21).
- **detach** — Detaches the Debugger from the program it is currently debugging (see “detach” on page 23).
- **eagleconfig** — Manages EAGLE and function instrumentation in INTEGRITY address spaces on a live target (see “eagleconfig” on page 23).
- **endtime** — Prints the elapsed time since the last **starttime** command (see “endtime” on page 26).
- **history** — Opens the **History** window, or clears the History log on the target (see “history” on page 26).
- **launcher** — Starts the Launcher (see “launcher” on page 27).
- **loadsym** — Loads additional debug symbols from the given file and merges them into the symbol table (see “loadsym” on page 27).

- **mev** — Requests the MULTI EventAnalyzer to perform the given operation (see “mev” on page 28).
- **monitor** — Saves the command list *commands* to send to the Debugger every time the process stops (see “monitor” on page 29).
- **mrw** — Launches or closes the MULTI ResourceAnalyzer (see “mrw” on page 30).
- **new** — Adds the given program to the target list and loads it in the Debugger window (see “new” on page 31).
- **output** — Redirects output normally sent to the Debugger into a file or to standard output (see “output” on page 32).
- **P** — Sends commands to processes and/or lists information about processes (see “P” on page 33).
- **q** — Prompts the user to quit MULTI (see “q” on page 35).
- **quit** — Closes the current Debugger window and exits MULTI if this is the last window (see “quit” on page 35).
- **quitall** — Exits MULTI (see “quitall” on page 36).
- **restore** — Restores the state of the Debugger from the specified file (see “restore” on page 37).
- **save** — Saves the state of the Debugger with respect to the currently selected task and target connection, if any (see “save” on page 37).
- **starttime** — Saves the current time for use with the **endtime** command (see “starttime” on page 38).
- **unloadsym** — Unloads from the symbol table all the debugging symbols for the specified file (see “unloadsym” on page 38).
- **wait** — Blocks command processing (see “wait” on page 38).

asm

asm [-v] [-replace *addr* [-force | -f]] [-opt "*options*"] *inst*

Select targets only

Assembles the given assembly instruction *inst* into the machine code it represents and prints this machine code. (If this feature is not supported for the current target,

an error message will be printed instead). This command takes the following arguments:

- `-v` — On targets where it is supported, prints any assembler command invoked by the Debugger, along with any output printed by the assembler command.
- `-replace addr` — Replaces the instruction at address *addr* with the assembled version of *inst* (if the assembling is successful). Note that this may fail if the instruction assembled has a different length than the instruction at address *addr*; see `-force` to avoid this.
- `-force` | `-f` — Only valid when `-replace` is used. This forces the replacement to proceed even if the instruction assembled has a different length than the instruction at address *addr*.
- `-opt "options"` — Assembles *inst* with the indicated assembler options. The quotation marks are required. See the documentation for your target's assembler for more information about possible options.



Note

Not all instructions are supported on all processor types. To get the correct assembly for some instructions, you may have to specify the processor in use on your target when using this command. For example, you may need to pass a **-cpu** option to the assembler:

```
asm -opt "-cpu ppc440gx" nop
```

attach

attach [-addressSpace *name*] *process_id* | *process_name* [-halt | -nohalt] [-exec *executable* | -noexec]

GUI only

Attaches to a process running on an RTOS, adds a new target list entry for the process, and loads the process in the Debugger window. Attaching to threads advertised by hardware targets is a deprecated use of this command.

You can use this command for native Linux/Solaris debugging. Note that unless you are logged in as `root`, you can only debug your own native processes.

Available options are:

- `-addressSpace name` — Specifies where the task (or process) is from on the INTEGRITY operating system. If `name` contains a space or other special characters, it should be enclosed in double quotation marks.
- `process_id` — Specifies the numeric ID of a task as displayed in the run-mode Task Manager, freeze-mode **OSA Explorer**, or native **Process Viewer** window.
- `process_name` — Specifies the name of a task as displayed in the run-mode Task Manager or freeze-mode **OSA Explorer** window. If `process_name` contains a space or other special characters, it should be enclosed in double quotation marks.
- `-halt` and `-nohalt` — Specify whether or not to halt the process on the target when the process is attached. If `-halt` or `-nohalt` is not specified, the behavior depends on the corresponding debug server's setting. Note that `-nohalt` is only supported with embedded debugging.
- `-exec executable` — Specifies the process's executable name.
- `-noexec` — Attempts to find the executable name for the process. If MULTI fails to find the name, it tries to attach to the process without an executable. While you can run, halt, and read and write to memory and registers without an executable, you cannot view source code or variables.
- Not specifying either `-exec executable` or `-noexec` — Attempts to find the executable name for the process. If MULTI fails to find the name, it asks you to select an executable. If you do not select one, MULTI tries to attach to the process without an executable, as with `-noexec`. The same limitations apply.

The **detach** command detaches from a process (see “detach” on page 23).

Corresponds to: **File** → **Attach to Process**

caches

caches [on | off]

GUI only

Enables or disables the use of both the memory cache (`_CACHE`) and the assembly cache (`_ASMCACHE`). For example, `caches on` is equivalent to `_CACHE = 1; _ASMCACHE = 1`. For more information about `_CACHE` and `_ASMCACHE`, see “System Variables” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

call

call *func* | *expr*

Prints the value of the function *func* or the expression *expr*. This command allows you to call:

- a function with the same name as a MULTI command or
- an expression that contains one or more variables with the same name as a MULTI command

This command does not use MULTI commands when resolving the name of the given function or expression.

dbnew

dbnew [c | n]

GUI only

Allows you to debug a different program. The **dbnew** command prompts you to select a new executable to debug. The selected executable is added to the target list and loaded into the Debugger window, replacing the previous program. If *c* is specified, the previous program is removed from the target list. If *n* is specified, the previous program is not removed from the target list and its status is unaffected. If you run this command without arguments, it has the same effect as `dbnew n`. See also “debug” on page 21.

Corresponds to: **File** → **Debug Program as New Entry**

debug

debug [*program_name*] [*core_file*]

Replaces your current program with a new program to debug given by *program_name*. If no new program is given, the current program's name is used. This command will have no effect unless the current process has executed. All monitors and monitor windows are deleted and any child processes of the current program are also killed.

The optional *core_file* argument applies only to Linux/Solaris. If *core_file* is specified, the Debugger will display the location of the program counter recorded in the core file. Otherwise, the Debugger will display the main routine. For more information, see Appendix I, “Debugging Linux/Solaris Core Files” in the *MULTI: Debugging* book.

The named files will be located using the default search path. See “Default Search Path for Files Specified in Commands” on page 14.

Corresponds to: 

Corresponds to: **File** → **Debug Program**

debugsnapshot

debugsnapshot [-noexecutables] [-unifiedloglimit=*num*] [-exclude_vas *vas_name* [-exclude_vas *vas_name*]...] [-nosource] *filename.dsnap*

debugsnapshot -load *filename.dsnap*

The first format of the **debugsnapshot** command saves a debug snapshot to a file on the host. This command is especially useful when run from a breakpoint in an error handler. This format is only supported with run-mode connections to INTEGRITY IoT (**rtserv2**). Additionally, synchronous debugging must be enabled (see “Stopping and Starting Tasks Synchronously” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in *MULTI: Debugging*).

The second format of the **debugsnapshot** command loads a saved debug snapshot.

Available options are:

- `-noexecutables` — Specifies that only target memory should be saved, not application binaries or debug information. A debug snapshot created with this option uses less disk space, but requires that programs it contains data about not be deleted, moved, or recompiled.
- `-unifiedloglimit=num` — Specifies that the debug snapshot will include only the most recent *num* bytes of data from the shared History and INTEGRITY TimeMachine log. A debug snapshot created with this option uses less disk space and takes less time to create, but does not provide access to older data. If you pass this option, try to choose a value for *num* that is small enough to minimize the size of your debug snapshot, but large enough to allow you to find the source of any failures. Note that you cannot exclude all log data by specifying `-unifiedloglimit=0`; zero in fact saves the entire log. Small values greater than zero are rounded up to a target-specific minimum log size.
- `-exclude_vas vas_name` — Excludes the address space *vas_name* from the debug snapshot. When an address space that is unrelated to the problem you are debugging uses a lot of memory, you can use this option to reduce the size of the debug snapshot. To exclude multiple address spaces, pass this option multiple times.
- `-nosource` — Excludes source code from the debug snapshot. A debug snapshot created with this option uses less disk space; however, if you want to see source when you load the saved debug snapshot, the Debugger must be able to locate the original, unmodified source files, or you must use the **sourceroot** command to load source files that you have saved yourself (see “sourceroot” on page 91).
- `filename.dsnap` — Specifies the full path of the file.
- `-load` — Loads the specified debug snapshot as a new entry in the Debugger window.

For more information, see Chapter 24, “Saving a Debug Snapshot” in *MULTI: Debugging*.

The first format of this command corresponds to **File → Save Debug Snapshot**.

The second format of this command corresponds to **File → Load Debug Snapshot**.

detach

detach [-run|-norun] [pr=*num*]

In run mode, detaches the Debugger from the current process and selects the next program in the target list. In a native environment, the entry for the current process is removed from the target list, but the debug server remains connected. All breakpoints are removed before detaching. See also “attach” on page 18.

- **-run** and **-norun** — Specify whether or not to resume the process on the target when the process is detached. If you do not specify one of these options, the behavior depends on the debug server, which usually resumes the process. (Note that the **detach** command may halt a currently running process before resuming it. This behavior depends on the debug server.)
- **pr=*num*** — Specifies that the process you want to detach from is placed in the Debugger's internal process slot number *num*. If no process slot is specified, the process on which the command is executed is detached.

Corresponds to: **File** → **Detach from Process**

eagleconfig

eagleconfig -enable | -disable | -revert | -list [-as *regex*] [-name *regex*]
[-fee | -eagle] [-verbose]

eagleconfig -save | -load *filename*

The first format of the **eagleconfig** command manages EAGLE and function instrumentation in INTEGRITY address spaces on a live target. As long as this instrumentation is available (this is dependent upon Builder options), you can use this command to enable or disable the instrumentation from MULTI, regardless of whether it was enabled or disabled at compile time. Furthermore, you can easily enable or disable instrumentation for a subset of functions or EAGLE tags. For example, you might want to disable instrumentation for a small function that is called many times, so as to reduce the performance impact of logging, or to fit more important data in the History log.

The second format of the **eagleconfig** command saves or loads the configuration of all instrumentation (whether enabled, disabled, or in the state specified at compile

time) for all address spaces. This is distinct from the save and restore that occurs on a per-address-space basis when the **AutoConfigEagle** configuration option is enabled (the default). For more information, see the description of this option in “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in *MULTI: Managing Projects and Configuring the IDE*.

The **-save** and **-list** options to this command are supported with debug snapshots and with run-mode connections to INTEGRITY IoT. All other options are only supported with run-mode connections to INTEGRITY IoT. This command may not be used while INTEGRITY TimeMachine is enabled.

If you are using the first format of this command, specify one of the following operations:

- `-enable` or `-disable` — If the system is paused, enables or disables instrumentation and clears the History log of all data, including data that has recently been decoded and data that is currently being decoded. Note that the History log may contain data used by History and INTEGRITY TimeMachine, and that clearing it means that you cannot run back in INTEGRITY TimeMachine or view data in History (with the exception that previously visible History data remains visible until you refresh the window).
- `-revert` — If the system is paused, reverts the state of instrumentation (enabled or disabled) back to that specified at compile time and clears the History log of all data, including data that has recently been decoded and data that is currently being decoded. See above for information about the consequences of clearing log data.
- `-list` — Prints the name and current state of instrumentation points.

The preceding operations apply to all instrumentation in the system unless you specify one or more of the following options:

- `-as regex` — Specifies that the preceding operations apply only to address spaces whose names match the regular expression *regex*.

Regular expression support is based on version 8.34 of the Perl Compatible Regular Expressions (PCRE) library package, which is open source software written by Philip Hazel and copyrighted by the University of Cambridge, England. It can be downloaded from the University of Cambridge FTP server [<ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>].

- `-name regex` — Specifies that the preceding operations apply only to the functions and EAGLE tags whose names match the regular expression *regex*.

Regular expression support is based on version 8.34 of the PCRE library package. For more information, see above.



Note

If the same function name or EAGLE tag name is used multiple times in the same address space, the instances cannot be modified separately.

- `-fee` or `-eagle` — Specifies that the preceding operations apply only to function entry/exit instrumentation or only to EAGLE instrumentation, respectively. By default, both types are affected.

You may optionally specify:

- `-verbose` — Prints instrumentation points that are modified by `-enable`, `-disable`, or `-revert` to the command pane.



Warning

Using this command to modify instrumentation state can lead to undefined contents in the History window. This occurs if the target is not reloaded prior to subsequent run-mode connections. For example, if you disable entry/exit instrumentation for some set of functions, then disconnect from the target, and later establish a new run-mode connection without restarting the target (so that the original modifications are still in place), the call stack for any affected address space may be incorrect in History. To mitigate this risk, disable the configuration option **assumeNoRuntimeEagleChanges** (see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in *MULTI: Managing Projects and Configuring the IDE*). However, note that doing so may result in a delay when you open History or run backwards for the first time after pausing the system.

endtime

endtime [-usec]

Prints the elapsed time since the last **starttime** command (see “starttime” on page 38). This allows you to measure the approximate amount of time between two events on the target. For example, when the target is halted, you might issue the following commands to save the current time, run the target, and then print the elapsed time when the target halts again:

```
starttime; cb; endtime
```

By default, **endtime** selects an appropriate time unit and then prints a decimal time such as 291.477808 sec, 101.601 msec, or 485 usec. If you pass the optional **-usec** argument, the elapsed number of microseconds are printed with no unit string. The previous examples would print as 291477808, 101601, and 485. This output is easier for a script to parse.



Note

The **endtime** command may overstate the time between two target events by up to a couple hundred milliseconds, depending on your target connection. Additionally, **endtime** may give slightly different values when measuring the same operation more than once, even if the actual time taken was the same. As a result, **starttime** and **endtime** are best used for rough measurements of at least a few hundred milliseconds.

history

history [-load *filename.hvs* | -invalidate]

Opens the **History** window, or clears the History log on the target, where:

- **-load *filename.hvs*** — Starts History on the saved History session file you specify. For more information, see “Saving and Loading History Sessions” in Chapter 23, “Viewing System Data with History” in *MULTI: Debugging*.
- **-invalidate** — Clears the History log on the target and does not open the History window. If you issue `history -invalidate` and then open the History window at a later time, the window only shows what happened on the

target after the command was executed. Note that the History log may contain data used by the History window and by INTEGRITY TimeMachine.

If you do not specify any arguments, the **History** window opens on data that has been generated for your system. For prerequisites, see “Getting Started with History” in Chapter 23, “Viewing System Data with History” in *MULTI: Debugging*.

Corresponds to: **TimeMachine** → **History**

launcher

launcher

Starts the Launcher.

Corresponds to: **Tools** → **Launcher**

loadsym

loadsym *filename* [-text *text_offset*] [-data *data_offset*] [-section "*section_name*" *section_offset*]...

Loads additional debug symbols from the file specified by *filename* and merges them into the symbol table. *filename* is searched for using the default search path (see “Default Search Path for Files Specified in Commands” on page 14). If the optional text and data offset values are supplied, the text and data addresses are offset by the given values. If the `-section` option is specified, addresses in the named section are offset by the given value. The `-section` option may be given multiple times to provide address offsets for different sections.

You can use this command in target environments where new code, typically position-independent code, is loaded to the target at run time. For example:

```
> loadsym a.out -text 0x20000
```

This command loads symbol information into the Debugger, but does not affect the target. You can load additional symbol information for a module during a debugging session.

**Warning**

Do not use this command to specify the primary executable that you want to debug. Instead, use the **debug** command followed by the **prepare_target** command (see “debug” on page 21 and “prepare_target” on page 253). For example, if you connect to a target on which your program is already loaded, first select **Direct hardware access** for the core that is running the program, and then issue the following commands:

```
> debug my_program
> prepare_target -verify=none
```

See also “unloadsym” on page 38.

mev

mev -close [*data_file*]

mev [-title *title_string*] [-noreload] [-raisewindow] *data_file*

mev [-title *title_string*] -newwindow [*data_file*]

mev [-oid *object_id*] -time *position_to_scroll_to* [-raisewindow] [*data_file*]

Requests the MULTI EventAnalyzer to perform the given operation, where:

- -close [*data_file*] — Closes the EventAnalyzer window displaying the data file *data_file*. If *data_file* is not specified, all EventAnalyzer windows are closed.
- *data_file* — Displays the specified data file, *data_file*, in an EventAnalyzer window. The following options can be placed before the *data_file* argument.
 - -title *title_string* — Specifies the title for the EventAnalyzer window displaying *data_file*.
 - -noreload — Indicates that the specified data file should not be reloaded if it is already being displayed in an EventAnalyzer window.
 - -raisewindow — Raises the existing EventAnalyzer window displaying the specified data file to the top.

- `-newwindow` — Opens a new EventAnalyzer window to display the specified data file. If you do not specify a data file, an empty EventAnalyzer opens. No more than one empty EventAnalyzer can be displayed at a time.
- `-time position_to_scroll_to` — Scrolls the EventAnalyzer window to the time specified by *position_to_scroll_to*. The following arguments can also be used with the `-time` option.
 - `-oid object_id` — Specifies the object ID, *object_id*, for the EventAnalyzer window to scroll to. This option should come before `-time`.
 - `-raisewindow` — Raises a previously launched EventAnalyzer window to the top.
 - *data_file* — Specifies which data file to display in the EventAnalyzer and scroll in. If no data file is specified, all EventAnalyzer windows are scrolled to the specified time on the specified object, if applicable.

monitor

monitor [0 | {*commands*} | *num* [{*commands*}]]

GUI only

Saves the command list *commands* to send to the Debugger every time the process stops. An unlimited number of monitors can be active at any time.

This command has five forms:

- **monitor** — Lists all of the monitors in order.
- **monitor** *num* — Deletes monitor number *num*. This does not renumber the current monitors. Thus, if you have four monitors and delete number 3, the remaining three will be numbered 1, 2, 4, creating an “empty slot” where 3 was formerly located.
- **monitor** { *commands* } — Inserts a monitor with the given command list in the first available empty slot.
- **monitor** *num* { *commands* } — Puts a monitor with the given command list in the *num* slot. It replaces any existing monitor in that position.
- **monitor** 0 — (The number zero.) Deletes all monitors.

See also “window” on page 304. For information about command lists, see “Using Command Lists in Debugger Commands” on page 12.

mrv

mrv [-title *string*] [-log *log_filename*] [-timeout *seconds*] [-port *port_num*] *target*

mrv -close [[-port *port_num*] *target*]

GUI only

Launches or closes the MULTI ResourceAnalyzer. The first format of the command launches the ResourceAnalyzer and the second format closes it.

Available options are:

- **-title *string*** — Specifies the string that appears in the title bar of the MULTI ResourceAnalyzer.
- **-log *log_filename*** — Specifies that messages from the target should be logged to *log_filename*.
- **-timeout *seconds*** — Specifies the timeout period (in seconds) for connecting to the target. If not specified, the host's default value is used.
- **-port *port_num*** — Specifies the port to connect to on *target*. If you do not specify a port, MULTI uses the default port used by the INTEGRITY target.
- ***target*** — Specifies the target you want to connect to.
- **-close [[-port *port_num*] *target*]** — Disconnects the MULTI ResourceAnalyzer from *port_num* on *target*. If you do not specify a port, MULTI disconnects from the default port used by the INTEGRITY target. If you do not specify *target*, the connection for the current ResourceAnalyzer is closed.

new

new *[[*-alias *component_alias*]*...* *[-bind component_alias]...program_name* *[pr=num]*
[core_file]

GUI only

Adds the program *program_name* to the target list and loads the program in the Debugger window.

Available options are:

- *-alias component_alias* — Adds the alias *component_alias* for the Debugger component *program_name* so that you can route commands to the component (see “route” on page 201). To list components and their aliases, use the **components** command (see “components” on page 110).
- *-bind component_alias* — Associates the program *program_name* with the CPU and connection represented by the given Debugger component. This argument is only supported if there is no pre-existing association between the connection and another executable. See also “Associating Your Executable with a Connection” in Chapter 7, “Preparing Your Target” in the *MULTI: Debugging* book.
- *program_name* — Adds the program *program_name* to the target list and loads the program in the Debugger window.
- *pr=num* — Specifies that the program *program_name* be placed in MULTI's internal program slot numbered *num*. If *num* is not specified, MULTI uses the first empty slot. If *num* is specified and that program slot is in use, the target list entry for that slot is reused to debug the new program. To see which program slots are in use, issue the **P** command (see “P” on page 33).
- *core_file* (Linux/Solaris only) — Specifies a core file for the program *program_name*. For more information, see Appendix I, “Debugging Linux/Solaris Core Files” in the *MULTI: Debugging* book.

If no arguments are specified, MULTI adds another instance of the current program to the target list and loads the program in the Debugger window.

output

output [-show] -server|-io|-multi [-prefix *prefix_string*|-noprefix] [-append] [-copytopane yes|no] [-silent] [*filename*|-normal]

Redirects output normally sent to the Debugger into a file or to standard output. You may optionally copy the redirected output to the Debugger.

When a string is redirected to a file or to standard output, a prefix is automatically printed before the text in order to distinguish it from different sources. You can change or disable the prefix with the `-prefix` or `-noprefix` options, respectively.

The options are:

- `-show` — Displays the current output settings.
- `-server` — Sets the destination for output normally sent to the target pane of the Debugger window (see “The Target Pane” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book).
- `-io` — Sets the destination for output normally sent to the I/O pane of the Debugger window (see “The I/O Pane” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book).



Note

Because I/O is handled by the underlying operating system in a native environment, you cannot use the **output** command to redirect output from a native process. On Linux/Solaris, the output is displayed in the terminal from which the Debugger was launched.

- `-multi` — Sets the destination for output normally sent to the command pane of the Debugger window (see “The Command Pane” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book).
- `-prefix` — Specifies the prefix string that is printed before the output.
- `-noprefix` — Indicates that no prefix is printed before the output.
- `-append` — Appends output to preexisting text in the specified destination file. If you do not specify this option, the output overwrites any preexisting text.
- `-copytopane yes` — Copies redirected output to the MULTI Debugger pane (target, I/O, or command) that serves as the default destination for the output

source (see the description of the `-normal` option below). This is the default behavior.

- `-copytopane no` — Does not copy redirected output to a MULTI Debugger pane.
- `-silent` — Does not print the destination of redirected output. If you do not specify this option, the new destination is printed to the command pane after you redirect the output.
- `filename` — Specifies the destination file for an output source.
- `-normal` — Resets the destination for the specified output source to its default destination, as listed below:
 - Debug server output is sent to the MULTI target pane.
 - Output from the process being debugged is sent to the MULTI I/O pane. (This does not apply if you are debugging in a native environment. See the description of `-io` earlier in this section.)
 - MULTI output is sent to the MULTI command pane.

If neither a filename nor the `-normal` option is specified, the output source is redirected to standard output.



Note

On Windows, MULTI Debugger output that is redirected to `stdout` via the **output** command is not usually visible unless you redirect `stdout`. For example, when launching the Debugger from the command line, you could redirect the Debugger's `stdout` to a file named **myout.txt**:

```
multi a.out > myout.txt
```

P

P `[[pr=num] subcommands]`

Sends *subcommands* to a process and then lists information about the process, or, if issued with no arguments, lists information (including process slot numbers) about all processes.

This command is used exclusively for multiprocess debugging, and most of the subcommands are used exclusively for native debugging.

If specified, the commands *subcommands* are sent to the process *num*, where *num* is MULTI's internal slot number for the process. For example, the command `P pr=1 b` toggles the state of the `b` flag in the process with slot number 1. If no process slot number is specified, the process currently being debugged is used. In either case, this command lists information about the process after sending *subcommands* to it.

Valid *subcommand* values for native debugging are:

- `b` — Toggles the flag causing breakpoint inheritance after forking. If the flag is set, children of the current process inherit all breakpoints set at the time of the fork.
- `c` — Toggles the flag causing children to be debugged. If the flag is set, children of the current process are added to the list of processes under the control of MULTI.
- `e` — Toggles the flag causing children to stop upon execution of the `exec()` system call. If the flag is set, this acts as if a breakpoint were encountered at the first instruction of routine `main()` in the exec'd program.
- `f` — Toggles the flag causing children to stop upon execution of the `fork()` system call. If the flag is set, this acts as if a breakpoint were encountered immediately following the fork.
- `i` — Toggles the flag that makes the child process inherit its settings from the parent. This will only have an effect if the `c` flag is also set.

After a `fork()` or `exec()` of a process, MULTI prints a message indicating that the process was stopped because either the `e` or `f` flag was set.

Valid *subcommand* values for native or embedded debugging are:

- `h` — Toggles the flag causing MULTI to halt a task when it is attached.
- `r` — Toggles the flag causing MULTI to resume a task when it is detached.

Valid *subcommand* values for INTEGRITY run-mode debugging are:

- `d` — Toggles the flag causing MULTI to debug a newly created task on target.

- `t` — Toggles the flag causing the target to stop a newly created task.

The following *subcommand* is deprecated in this version and remains for compatibility. The **signal** command should be used instead (see “signal” on page 176).

- `s num` (Linux/Solaris only) — Sends signal *num* to the current process.

q

q

Non-GUI only

Prompts the user to quit MULTI. This command works only in non-GUI mode. When prompted, your choices are as follows.

- `n` — Cancels the quit request. This is the default choice.
- `s` — Saves breakpoints and the directory list to the file named **multistate**, and then exits MULTI.
- `y` — Exits MULTI.

quit

quit [*all*] [*this*] [*ask* | *confirm* | *auto* | *force*] [*window* | *entry*]

Closes the current Debugger window or removes the current target list entry. If this is the last entry, the Debugger exits. If you do not specify any arguments and a process has started, MULTI prompts you to kill the process or, if the target allows, detaches the Debugger from the process and lets it run. If you do not specify any arguments and no process has started, the Debugger closes without prompting you.

You can specify one of the following arguments to modify the behavior of this command:

- `all` — Is equivalent to the **quitall** command except that MULTI prompts you before killing an active process or, if the target allows, detaches the Debugger from the process and lets the process run (see “quitall” on page 36).
- `this` — Closes the current target list entry. MULTI prompts you first.

- `ask` — Uses the **promptQuitDebugger** configuration option to determine whether or not to prompt you before quitting the Debugger. The default setting is to prompt only if a process has started. See “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.
- `confirm` — Always prompts you before quitting the Debugger.
- `auto` — Quits the current Debugger window. If a process has started, MULTI prompts you to kill the process or, if the target allows, detaches the Debugger from the process and lets it run. If no process has started, MULTI closes the Debugger without prompting you.
- `force` — Closes the current Debugger window and kills the current process without prompting.
- `window` — Closes the current Debugger window, but does not detach from or terminate any process unless the current Debugger window is the only one remaining. If the current Debugger window is the only one remaining, MULTI detaches from run-mode processes and terminates non-run-mode processes when it closes the window. This option corresponds to  and **File → Close Debugger Window**.
- `entry` — Removes the currently selected entry from the target list. In run mode, MULTI detaches from the process. If you are not debugging in run mode, MULTI terminates the process. If the entry is the only remaining entry in the target list, all Debugger windows close and the Debugger exits. This option corresponds to **File → Close Entry**.

quitall

quitall

Causes MULTI to exit. This closes all of MULTI's windows. If you have edits that have not been saved, you will be given the chance to save them before exiting. The Debugger does not prompt you before killing active processes.

Corresponds to: **File → Exit All**

restore

restore [*filename*]

GUI only

Restores the state of the Debugger from *filename* and, if it exists, from *filename.bps*. If you do not specify *filename*, this command restores the state of the Debugger from **multistate** and, if it exists, from **multistate.bps**. These files must have been created with the **save** command (see “save” on page 37) or with **Config** → **State** → **Save State**. If you were connected to a debug server when you saved, the **restore** command also attempts to reconnect you to that debug server.

MULTI searches for files using the default search path. See “Default Search Path for Files Specified in Commands” on page 14.



Note

This command may not be able to restore group breakpoints.

Corresponds to: **Config** → **State** → **Restore State**

save

save [*filename*]

Saves the state of the Debugger with respect to the currently selected task and target connection, if any. This writes out source directories as set by the **source** command (see “source” on page 90), and the current target connection, if any, to the file *filename* or to the file **multistate** if no filename is specified on the command line. If breakpoints exist, they are written out to *filename.bps* or to **multistate.bps**. These files may later be used by the **restore** command to restore the state of the Debugger (see “restore” on page 37). Note that the **restore** command may not be able to restore group breakpoints.



Note

If your target supports it, consider saving a debug snapshot instead. See Chapter 24, “Saving a Debug Snapshot” in *MULTI: Debugging*.

Corresponds to: **Config** → **State** → **Save State**

starttime

starttime

Saves the current time. When used in conjunction with the **endtime** command, this allows you to measure the approximate amount of time that has elapsed between two events on the target. For more information, see “endtime” on page 26.

unloadsym

unloadsym *filename*

Unloads from the symbol table all of the debugging symbols for the file *filename*.

You can use this command with the **loadsym** command to load debugging information for a stripped executable at run time (see “loadsym” on page 27).

For example:

```
> debug a.out.stripped
> loadsym a.out.debug
> unloadsym a.out.stripped
```

The filename will be searched for using the default search path. See “Default Search Path for Files Specified in Commands” on page 14.

wait

wait [-global] [-time *milliseconds* [-timeoutcmd {*commands*}]] [-search] [-pause] [-stop] [[-not] -multiStatus *status_name_or_value*] [-py [-all | -cmd | -pane]] [[-show | -goaway] [-taskStack | -taskStatus *status*] [-addressSpace *addressSpace_name*] -taskName *task_name* | -taskid *task_id*] [-debugsnapshot] [-load] [-unload] [-runmode_partner]

Blocks command processing. While command processing is blocked, MULTI cannot accept new commands. However, it can still refresh its windows and handle target events. To abort the **wait** command, press the **Esc** key.

The available arguments are:

- `-global` — Blocks command processing for all MULTI processes in the same MULTI session. If you do not specify `-global`, only the current process is blocked. This argument may be used in conjunction with any other argument.
- `-time milliseconds` — Blocks command processing for *milliseconds*. If used in conjunction with another argument, *milliseconds* represents the maximum period of time that command processing is blocked, whether or not the action specified by the second argument has occurred.

For example, the following command blocks command processing for up to 2 seconds while waiting for the current process to halt:

```
> wait -time 2000 -stop
```

You may additionally specify `-timeoutcmd {commands}` after `-time milliseconds` and in conjunction with a third argument. If the timeout occurs before the action specified by the third argument, *commands* are executed. For example, the following command blocks command processing for up to 2 seconds while waiting for the current process to halt. If the current process does not halt in 2 seconds, this command quits the current process:

```
> wait -time 2000 -timeoutcmd {quit} -stop
```

For information about specifying a list of commands with `-timeoutcmd`, see the description of *{commands}* in “Debugger Command Conventions” on page 3.

- `-search` — Blocks command processing until the search in progress (if any) has finished.
- `-pause` — Blocks command processing until the system is paused. The status shown in the Debugger target list for the currently selected task may be **Paused by halted_task/core** or **Stopped**. This option is only applicable when synchronous debugging is enabled. For information about synchronous debugging in a run-mode environment, see “Stopping and Starting Tasks Synchronously” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in *MULTI: Debugging*. For information about synchronous debugging in a freeze-mode environment, see “Synchronous Debugging on a Multi-Core Target” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in *MULTI: Debugging*.

- `-stop` — Blocks command processing until the current process halts. This is the default behavior if you do not specify any argument to the **wait** command. It is also a special case of the `-multiStatus` syntax, described next.
- `[-not] -multiStatus status_name_or_value` — Blocks command processing until the MULTI process enters the specified status or, if `-not` is specified, until the process exits the specified status. Status name/value possibilities follow.

Status Name	Status Value
Nil	1
Stopped	2
Running	3
Dying	4
Fork	5
Exec	6
Continue	7
Zombie	8
targetFrozen	N/A

The status name `targetFrozen` indicates the target's system halted status. The other names are normal statuses for MULTI processes, and the numeric values are MULTI-internal values. For information about accessing the current process status, see the `_STATE` variable in “System Variables” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

- `-py [-all|-cmd|-pane]` — Blocks command processing until pending Python statements complete. Use the following optional arguments to specify the source of the pending Python statements:
 - `-all` — Indicates that the Python statements may come from any source. This is the default.
 - `-cmd` — Indicates that the Python statements come from the **python** or **py** Debugger command. For information about these commands, see “python, py” on page 222.
 - `-pane` — Indicates that the Python statements come from the Debugger's Python pane.

- `[-show] -taskStack [-addressSpace addressSpace_name] (-taskName task_name) | (-taskid task_id)` (The parentheses are used for grouping) — Blocks command processing until stack information for `task_name/task_id` located in `addressSpace_name` appears in the target list and Task Manager. If the task name is unique, you do not need to specify an address space.

`-goaway` is not generally useful with `-taskStack`. `-show` is the default.

- `[-show] -taskStatus status [-addressSpace addressSpace_name] (-taskName task_name) | (-taskid task_id)` — Blocks command processing until `task_name/task_id` located in `addressSpace_name` attains the specified status in the target list and Task Manager. Status names vary by RTOS. If you are using a run-mode connection to debug INTEGRITY, status names are: Ready, SUSPENDED, Pended, Running, Created, HostIO, Zombied, Halted, DebugBrk, SysHalt, GrpHalt, Checkpoint, and Incipient. If you are debugging a non-INTEGRITY RTOS and `status` contains a space, you must enclose it with quotation marks.

`-goaway` is not generally useful with `-taskStatus`. `-show` is the default.

- `[-show|-goaway] [-addressSpace addressSpace_name] (-taskName task_name) | (-taskid task_id)` — Blocks command processing until `task_name/task_id` located in `addressSpace_name` appears in the target list and Task Manager (`-show`) or disappears from the target list and Task Manager (`-goaway`). `-show` is the default.
- `-debugsnapshot` — If you are saving a debug snapshot, blocks command processing until the save is complete.
- `-load` — Blocks command processing until all dynamic downloads for the current connection are complete (regardless of the success or failure of the downloads).
- `-unload` — Blocks command processing until the last module to be unloaded is fully unloaded from the target.
- `-runmode_partner` — Blocks command processing until the freeze-mode connection's run-mode partner has been created and initialized. See “Automatically Establishing Run-Mode Connections” in Chapter 4, “INDRT2 (rtserv2) Connections” in the *MULTI: Debugging* book.

For more information about the Task Manager window, see Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

The **wait** command can be useful when scripting test cases or in a situation like the following. Suppose you are debugging in run-mode and want to attach to a task named **My Task** when it appears in the Task Manager window. You can enter:

```
> wait -taskname "My Task"; attach "My Task"
```

Chapter 3

Breakpoint Command Reference

Contents

Breakpoint Commands	45
---------------------------	----

The Debugger provides a variety of commands for setting and removing breakpoints. Many of the breakpoint commands have a similar syntax, since all breakpoints have similar attributes.

Every breakpoint must be associated with an address. Most of the breakpoint commands take an optional address expression that specifies the location of the breakpoint. (For more information about the *address_expression* argument, see “Using Address Expressions in Debugger Commands” on page 5.) If an address is not specified, the current line is used. In GUI mode, the current line is indicated by the blue arrow.

Some of the breakpoint commands allow you to specify whether the breakpoint is an any-task breakpoint or a task-specific breakpoint. An any-task breakpoint can be hit by any task in the AddressSpace that the breakpoint is set in, while a task-specific breakpoint can only be hit by the task it is set on. The */at* (any-task) and */tt* (task-specific) options are only meaningful if you are using INTEGRITY or another operating system that supports them.

A breakpoint can have a breakpoint count, which causes the breakpoint to be skipped a specified number of times before it stops the process. The breakpoint count is decremented by 1 each time the breakpoint is skipped. When the breakpoint count reaches 1, the breakpoint stops the process (and continues to stop the process every subsequent time it is reached). To set a breakpoint count, add the argument *@bp_count* before the command list. For example, the following command sets a breakpoint with a count of 4:

```
> b @4
```

In this example, the breakpoint will be hit the fourth time the line of code is executed, and will continue to be hit every subsequent time that line is executed.

All breakpoint commands that contain the argument *{commands}* can take a list of commands. The commands in the list are executed when the scripted breakpoint is hit. For more information, see “Using Command Lists in Debugger Commands” on page 12.

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

For information about setting breakpoints using the GUI, see “Using Breakpoints and Tracepoints” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

Breakpoint Commands

The following list provides a brief description of each breakpoint command. For a command's arguments and for more information, see the page referenced.

- **b** — Sets a breakpoint at the specified location (see “b” on page 46).
- **B** — Lists information about breakpoints (see “B” on page 48).
- **bA** — Sets a temporary breakpoint (see “bA” on page 49).
- **bi, bI** — Sets a permanent or temporary breakpoint in a function prologue (see “bi, bI” on page 49).
- **bif** — Sets a conditional breakpoint that will stop the process if the given condition evaluates to true (see “bif” on page 50).
- **bpload** — Loads breakpoints from the specified file (see “bpload” on page 51).
- **bpsave** — Saves breakpoints to the specified file (see “bpsave” on page 51).
- **bpview, breakpoints** — Opens the **Breakpoints** window (see “bpview, breakpoints” on page 52).
- **bt** — Displays a message every time the specified function is entered or exits (see “bt” on page 52).
- **bu, bU** — Sets a permanent or temporary up-level breakpoint (see “bu, bU” on page 53).
- **bx, bX** — Sets a permanent or temporary breakpoint at the exit point of a function (see “bx, bX” on page 54).
- **d** — Deletes the software breakpoint at the specified address expression or with the given label (see “d” on page 55).
- **D** — Deletes software breakpoints (see “D” on page 56).
- **dz** — Allows you to restore, view, or permanently remove previously deleted breakpoints (see “dz” on page 56).
- **edithwbp** — Opens a dialog box that you can use to edit the hardware breakpoint at the current source line (see “edithwbp” on page 58).

- **editswbp** — Opens a dialog box that you can use to edit the software breakpoint at the current source line (see “editswbp” on page 58).
- **hardbrk** — Lists, deletes, or sets hardware breakpoints (see “hardbrk” on page 59).
- **install_bp_on_request** — Controls when the Debugger installs breakpoints on the target (see “install_bp_on_request” on page 64).
- **rominithbp** — Sets or removes the post-initialization hardware breakpoint used in ROM and ROM-to-RAM copy debugging (see “rominithbp” on page 65).
- **sb** — Sets target-specific breakpoints (see “sb” on page 66).
- **setbrk** — Sets a new breakpoint or removes an existing breakpoint at the current line or at the current address (see “setbrk” on page 67).
- **sethbp** — Sets a hardware execute breakpoint at the current location in your program code (see “sethbp” on page 68).
- **stopif** — Sets a conditional breakpoint at the line number specified (see “stopif” on page 68).
- **stopifi** — Sets a conditional breakpoint on the machine instruction at the specified address (see “stopifi” on page 69).
- **tog** — Toggles the active status of the given address expression or breakpoints (see “tog” on page 70).
- **Tog** — Toggles the active status of all breakpoints (see “Tog” on page 71).
- **watchpoint** — Sets a watchpoint on the address indicated, which causes the process to halt when the address is written to (see “watchpoint” on page 71).

b

b [%bp_label] [@bp_count] [&] [/s] [/off] [/at | /tt | /type_gt [@task_group] [/trigger @task_group]] [address_expression] [{commands}]

Sets a breakpoint at the specified location, where:

- %bp_label — Specifies a breakpoint label (see “Breakpoint IDs and Labels” on page 10).
- @bp_count — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).

- `&` — Causes the breakpoint to beep when it is hit.
- `/s` — Suppresses messages when the breakpoint is hit.
- `/off` — Sets an inactive breakpoint. This allows you to set a breakpoint on a running target without halting the target.
- `/at` — Sets an any-task breakpoint. This is the default on targets that support any-task breakpoints.
- `/tt` — Sets a task-specific breakpoint. This is the default on targets that do not support any-task breakpoints.
- `/type_gt [@task_group]` — Sets a group breakpoint for the specified task group. Any task in the task group can hit the breakpoint. If `@task_group` is not present, MULTI uses a temporary group that contains only the task itself to create the group breakpoint. If `task_group` contains spaces, enclose it with double quotation marks. The `{commands}` and `/type_gt [@task_group]` arguments are mutually exclusive. The breakpoint count (`@bp_count`) of group breakpoints must be 1 (default). See “Setting Breakpoints for Task Groups” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.
- `/trigger @task_group` — Specifies the task group that stops when a group breakpoint is hit. All tasks in the task group stop along with the task that hits the breakpoint. If `task_group` contains spaces, enclose it with double quotation marks. See “Setting Breakpoints for Task Groups” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

This option is only applicable if `/type_gt` is specified.

- `address_expression` — Specifies an address (see “Using Address Expressions in Debugger Commands” on page 5). In GUI mode, if `address_expression` is ambiguous, the **b** command opens a dialog box listing all functions that match the wildcard pattern `address_expression`. You can pick some, all, or none of the functions listed. For more information about the dialog box, see “Function Ambiguities and the Browse Dialog Box” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book. If you do not specify an address expression, the breakpoint is set at the location of the blue current line pointer (➡).
- `{commands}` — Specifies a list of commands to be performed (see “Using Command Lists in Debugger Commands” on page 12). The `/type_gt [@task_group]` and `{commands}` arguments are mutually exclusive.

The options you specify determine what breakpoint marker is displayed on the left side of the source pane. See “Breakdots, Breakpoint Markers, and Tracepoint Markers” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

If a function name is specified (for example with the command `b Fly`), the breakpoint is not set at the first machine instruction of the function, but rather at the first machine instruction after the function's stack setup code, if any exists. This ensures that the arguments and local variables of a function are read correctly when you stop in that function. (See “The Call Stack Window and Function Prologues and Epilogues” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.) For information about stopping at the first machine instruction of a function, see “bi, bI” on page 49.

B

B [*address_expression* | *breakpoint_list*]

Lists information about breakpoints. The information is listed in the following order:

ID bp_label location address count: flags commands

If no argument is specified, this command lists information about all breakpoints. You can also specify an address or a list of breakpoints. For more information, see “Using Address Expressions in Debugger Commands” on page 5 and “Breakpoint Ranges and Lists” on page 11.

Example:

```
> b main#5
> b %my_bp_name main#6 { print "Here I am"; }
> b main#7 { print "main#7" }
> tog main#5
> B
0          main#5: 0x10204 count: 1 (inactive)
1 my_bp_name main#6: 0x01220 count: 1 <{ print "Here I am"; }>
2          main#7: 0x1022c count: 1 <{ print "main#7" }>
> B %my_bp_name:%2
1 my_bp_name main#6: 0x10220 count: 1 <{ print "Here I am"; }>
2          main#7: 0x1022c count: 1 <{ print "main#7" }>
```

**Note**

Entering the **l b** (lowercase **l** lowercase **B**) command is the same as entering **B** in the command pane (see “l” on page 115).

**Note**

You can list information about deleted breakpoints by using the **dz -list** command (see “dz” on page 56).

bA

bA [%bp_label] [@bp_count] [&] [/s] [/at | /tt] [address_expression] [{commands}]

Sets a temporary breakpoint using many of the same options as the **b** command (see “b” on page 46). A temporary breakpoint is automatically removed when it is hit.

bi, bI

bi [%bp_label] [@bp_count] [/at | /tt] [address_expression] [{commands}]

bI [%bp_label] [@bp_count] [/at | /tt] [address_expression] [{commands}]

(The second command contains an uppercase **i**.)

Sets a breakpoint in a function prologue. (The **bI** command sets a temporary breakpoint; the **bi** command sets a permanent breakpoint.)

- %bp_label — Specifies a breakpoint label (see “Breakpoint IDs and Labels” on page 10).
- @bp_count — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- /at — Sets an any-task breakpoint. This is the default on targets that support any-task breakpoints.
- /tt — Sets a task-specific breakpoint. This is the default on targets that do not support any-task breakpoints.
- address_expression — Specifies a location (see “Using Address Expressions in Debugger Commands” on page 5). If the location is specified

as a function name, the breakpoint is set at the beginning of the function prologue, if any. (Other breakpoint commands would set the breakpoint at the first instruction after the prologue.) If the location is not a function name, the breakpoint is set at that location.

- *{ commands }* — Specifies a list of commands to be performed (see “Using Command Lists in Debugger Commands” on page 12).

bif

bif [%bp_label] [@bp_count] [/at | /tt] *address_expression condition*

Sets a conditional breakpoint that will stop the process if *condition* evaluates to true.

- %bp_label — Specifies a breakpoint label (see “Breakpoint IDs and Labels” on page 10).
- @bp_count — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- /at — Sets an any-task breakpoint. This is the default on targets that support any-task breakpoints.
- /tt — Sets a task-specific breakpoint. This is the default on targets that do not support any-task breakpoints.
- *address_expression* — See “Using Address Expressions in Debugger Commands” on page 5.
- *condition* — Expression in the current language.

For example:

```
> bif funcname {i != 0}
Stop if i != 0 set at 0x20e6b8.
```

will stop the process when the function *funcname* is entered, if the variable *i* is nonzero.

See also “stopif” on page 68.

bpload

bpload *filename*

Loads breakpoints from *filename*, where *filename* is a file created with **bpsave** (see “bpsave” on page 51).

You can also use the **Load** button in the **Breakpoints** window to achieve the same results.



Note

This command and button may not be able to correctly load group breakpoints.

bpsave

bpsave *filename* [*breakpoint_list*]

Saves *breakpoint_list* to *filename*. If *breakpoint_list* is not specified, all breakpoints are saved. The breakpoints are generally preserved in the form *file#func#line* to provide maximal portability between debugging sessions.

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

```
> bpsave brkpts.lst
```

This saves the breakpoints to the file **brkpts.lst**. (You can also use the **Save** button in the **Breakpoints** window to achieve the same results.)

Later, when you restart the Debugger, you can issue the following command:

```
> bpload brkpts.lst
```

This restores the breakpoints from the previous debugging session.

See also “bpload” on page 51.

bpview, breakpoints

bpview

breakpoints

GUI only

Opens the **Breakpoints** window, which you can use to add, change, or delete software breakpoints, hardware breakpoints, and tracepoints. See “Viewing Breakpoint and Tracepoint Information” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

Corresponds to: 

Corresponds to: **View** → **Breakpoints**

bt

bt [*@bp_count*] [/at | /tt] *func_name*

Displays a message every time the specified function is entered or exits, where:

- *@bp_count* — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- /at — Sets an any-task breakpoint. This is the default on targets that support any-task breakpoints.
- /tt — Sets a task-specific breakpoint. This is the default on targets that do not support any-task breakpoints.
- *func_name* — Specifies a function name.

The displayed message states whether the function is entered or exited. The message printed on exit also provides the return value of the specified function.



Note

This command may produce unexpected behavior if your compiler optimization settings result in *func_name* having multiple exit points or if the exit point cannot be found.

bu, bU

bu [*@bp_count*] [*stacklevel*] [{*commands*}]

bU [*@bp_count*] [*stacklevel*] [{*commands*}]

Sets an up-level breakpoint. The breakpoint is permanent if you use **bu** and temporary if you use **bU**.

- *@bp_count* — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- *stacklevel* — Call stack level.
- {*commands*} — See “Using Command Lists in Debugger Commands” on page 12.

The breakpoint is set immediately after the return to the level specified by *stacklevel*. The *stacklevel* is specified with a numeric value without an underscore (for example, 5; see Chapter 5, “Call Stack Command Reference” on page 77). If *stacklevel* is not specified, the breakpoint is set one level up from the current function. For example, a way to recover after accidentally single-stepping into a function is:

```
> bU; c
```

to set a temporary, up-level breakpoint and continue.

The **bu** and **bU** commands rely on the Debugger's ability to generate a partial stack trace. They may not work correctly (for example, they may set a breakpoint at the wrong address) if the stack trace obtained by the Debugger is incorrect. For restrictions on tracing the call stack, see “Viewing Call Stacks” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

See also “cu, cU” on page 180.

bx, bX

bx [%bp_label] [@bp_count] [/at | /tt] [address_expression] [{commands}]

bX [%bp_label] [@bp_count] [/at | /tt] [address_expression] [{commands}]

Sets a breakpoint at the exit point of a function. The breakpoint is permanent if you use **bx** and temporary if you use **bX**.

- *%bp_label* — Specifies a breakpoint label (see “Breakpoint IDs and Labels” on page 10).
- *@bp_count* — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- */at* — Sets an any-task breakpoint. This is the default on targets that support any-task breakpoints.
- */tt* — Sets a task-specific breakpoint. This is the default on targets that do not support any-task breakpoints.
- *address_expression* — If a call stack level is specified in the address expression, this command sets a breakpoint at the exit point of the function at the specified stack level. See Chapter 5, “Call Stack Command Reference” on page 77. If a function name is specified as the address expression, this command sets a breakpoint at the exit point of the function. For more information about address expressions, see “Using Address Expressions in Debugger Commands” on page 5.
- *{commands}* — The specified *commands* are executed when the breakpoint is hit. For more information, see “Using Command Lists in Debugger Commands” on page 12.

If no arguments are specified, this command sets a breakpoint at the exit point of the current function. The exit point is where all returns from this function will go through.

The following are two examples of this command:

```
> bx foo
```

```
> bx "foo.c"#a_routine
```

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



Note

This command may produce unexpected behavior if your compiler optimization settings result in the function having multiple exit points or if the exit point cannot be found.

d

d [[/force] *address_expression* | *breakpoint_list* | %*bp_label*]

Deletes one or more software breakpoints at the specified *address_expression*, in the specified *breakpoint_list*, or with the specified *bp_label*, where:

- */force* — Removes all software breakpoints that match *address_expression*.
- *address_expression* — Specifies the address (see “Using Address Expressions in Debugger Commands” on page 5). In GUI mode, if *address_expression* is ambiguous and */force* is not specified, the **d** command opens a dialog box listing all software breakpoints in functions that match the wildcard pattern *address_expression*. You can delete some, all, or none of the software breakpoints listed.
- *breakpoint_list* — Specifies a list of software breakpoints. For more information, see “Breakpoint Ranges and Lists” on page 11.
- %*bp_label* — Specifies a software breakpoint label (see “Breakpoint IDs and Labels” on page 10).

If no arguments are given, all software breakpoints at the current line are removed.

This command removes software breakpoints and all of their associated attributes. If you want to temporarily disable a breakpoint without deleting it, use the **tog** command (see “tog” on page 70).

D

D [/at | /tt]

Deletes software breakpoints:

- If the OSA master process is selected in the target list — Deletes all normal software breakpoints (except group breakpoints) in the master process. For more information, see “Working with Freeze-Mode Breakpoints” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book.
- If a run-mode AddressSpace is selected in the target list and you are using INTEGRITY IoT or INTEGRITY version 10 or later — Deletes all any-task breakpoints in the AddressSpace.
- If a task is selected in the target list and you specify /at (the default) — Deletes all task-specific and any-task software breakpoints in the task.
- If a task is selected in the target list and you specify /tt — Deletes all task-specific software breakpoints, but not any-task software breakpoints, in the task.

The /at and /tt options are only meaningful when a task is selected in the target list.

Corresponds to: **Debug → Remove All Breakpoints**

dz

dz [soft | hard | sobp] [bp_ID_list]

dz -gui [soft | hard | sobp]

dz -line [list] soft | hard

dz -list [all] [soft | hard | sobp]

dz -clear [soft | hard | sobp] [bp_ID_list]

The first format of the **dz** command restores breakpoints deleted in the last breakpoint deletion operation or, if `bp_ID_list` is specified, the breakpoints whose IDs are listed.

The second format of the **dz** command opens the **Breakpoints Restore** window, which allows you to view and restore deleted breakpoints.

The third format of this command restores breakpoints deleted from the current line's last breakpoint deletion operation, or, if `list` is specified, it lists the breakpoints that were last deleted from the current line.

The fourth format prints a list of breakpoints that have been deleted and that can be restored.

The fifth format removes breakpoints permanently.

The following options can be used with the **dz** command:

- `-gui` — Opens the **Breakpoints Restore** window, from which you can view and restore deleted breakpoints. If specified in conjunction with `soft`, `hard`, or `sobp`, the window opens on the **Software** tab (the default), the **Hardware** tab, or the **Shared Object** tab, respectively. For information about the **Breakpoints Restore** window, see “The Breakpoints Restore Window” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.
- `-line` — Restores the software or hardware breakpoint(s) last deleted from the line where the current line pointer (➡) is located, or, if specified in conjunction with `list`, lists the software or hardware breakpoint(s) last deleted from the line where the current line pointer is located but does not restore them.
- `-list` and `list` — Prints a list of breakpoints that have been deleted and that can be restored. If specified in conjunction with the `all` option, all restorable, deleted breakpoints are printed to the command pane. Otherwise, only those removed in the last deletion operation are printed to the command pane.
- `-clear` — Permanently removes all applicable breakpoints. For example, if specified in conjunction with the `soft` option, all previously deleted software breakpoints (not only those deleted in the last deletion operation) are permanently deleted and can no longer be restored. This option is useful if you want to remove one or more breakpoints from the list of deleted breakpoints (see the `-list` and `list` options).

- `soft` — Applies the specified operation only to software breakpoints.
- `hard` — Applies the specified operation only to hardware breakpoints.
- `sobp` — Applies the specified operation only to shared object breakpoints.
- `all` — Applies the specified operation to all deleted breakpoints, not just to those from the last breakpoint deletion operation.
- `bp_ID_list` — Applies the specified operation only to those deleted breakpoints whose ID numbers are contained in this space-delimited list. When you run the **dz -list** command, the breakpoint ID numbers of deleted breakpoints precede the breakpoint descriptions. For more information about breakpoint IDs, see “Breakpoint IDs and Labels” on page 10.

edithwbp

edithwbp

GUI only

Opens a dialog box that you can use to edit the hardware breakpoint at the current source line. If no hardware breakpoint is set at the current line (as indicated by the blue context arrow), you can use the dialog box to create a new hardware breakpoint. If hardware breakpoints are not supported on the currently connected target, you will not be able to set the breakpoint. See “Creating and Editing Hardware Breakpoints” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

editswbp

editswbp

GUI only

Opens a dialog box that you can use to edit the software breakpoint at the current source line. If no software breakpoint is set at the current line (as indicated by the blue context arrow), you can use the dialog box to create a new software breakpoint. See “Creating and Editing Software Breakpoints” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

hardbrk

hardbrk [at]

hardbrk [at] delete= *num* | * | *expr*

hardbrk [enabled | disabled] [rolling] [@bp_count] [read] [write] [execute]
[mask=*mask*] [data=*data_value* [dmask=*data_mask*]] [vm] [at] [tm | live] [--] *expr*
[:size] [{*commands*}]

The first format of the **hardbrk** command lists all currently set hardware breakpoints, or, if *at* is specified, it lists all any-task hardware breakpoints or any-core hardware breakpoints, depending on your environment. Each breakpoint listed is allocated an identifying number.

The second format of the **hardbrk** command deletes one or more hardware breakpoints.

The third format of the **hardbrk** command sets a hardware breakpoint. If no slots remain on the target for a new hardware breakpoint, MULTI first disables an existing hardware breakpoint that was set with the rolling attribute (if any), and then it sets the hardware breakpoint. If no rolling hardware breakpoints are set on the target and no slots are available, a hardware breakpoint is not set.

All formats of this command require that you pass the options in the order they are presented above.



Note

Hardware breakpoints are implemented through direct hardware support. Only a limited number of hardware breakpoints (usually fewer than four) may be set on targets that support them.



Note

Even targets and debugging environments that allow you to set hardware breakpoints may not support all the breakpoint capabilities described here; neither do all targets support the precise semantics described. For caveats and limitations related to using hardware breakpoints in specific debugging environments, such as in TimeMachine, in INTEGRITY TimeMachine, or with INTEGRITY, see “Hardware Breakpoints in Specific Environments” in Appendix H, “Advanced Breakpoint Topics”

in *MULTI: Debugging*. For information about how your target handles hardware breakpoints if you are using a Green Hills Probe or SuperTrace Probe, see the documentation about target-specific hardware breakpoint support in the *Green Hills Debug Probes User's Guide*.

The following options can be used with this command:

- `at` — Specifies that the breakpoint is an any-task or any-core hardware breakpoint, as follows:
 - If you are using a run-mode connection to debug INTEGRITY IoT, INTEGRITY version 10 or later, or VxWorks, this option specifies that the breakpoint is an any-task hardware breakpoint. Normally, a hardware breakpoint can only be hit by the task in which it is set. An any-task hardware breakpoint, on the other hand, can be hit by any task in the address space in which it is set.
 - If you enabled synchronous debugging on your multi-core target, this option specifies that the breakpoint is an any-core hardware breakpoint. An any-core hardware breakpoint can be hit by any core in the multi-core system.

If this option is the only one specified (that is, `hardbrk at`), it lists all any-task or any-core hardware breakpoints.

If this option is passed before the `delete` option, it deletes the any-task/any-core hardware breakpoint(s) specified by `num` or `expr`. It has no effect on `*`.

- `delete=num` — Deletes the hardware breakpoint numbered `num`. To list the number associated with each hardware breakpoint, enter **hardbrk** with no arguments.
- `delete=*` — Deletes all hardware breakpoints.
- `delete=expr` — Deletes the breakpoint at the specified location. `expr` can be a memory address, variable, or pointer name. If you specify a memory address using a numerical value, the Debugger first attempts to interpret it as a breakpoint number (`delete=num`). If no hardware breakpoint of that number exists, the Debugger treats the value as an address.

If you do not specify `expr`, `num`, or `*` with the `delete=` option, the current line is used.

- `enabled` and `disabled` — Specify that the hardware breakpoint should initially be enabled or disabled. A disabled hardware breakpoint does not consume any target resources. The default is `enabled`.
- `rolling` — Indicates that the hardware breakpoint can be automatically disabled to open a slot when you want to set a new hardware breakpoint in the future.
- `@bp_count` — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- One or more of the following attributes may be specified. Do not specify `read` and/or `write` in conjunction with `execute`.
 - `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.

The default attribute for an address in a `text` section is `execute`. The defaults for `data` and other sections are `read` and `write`. Often the break only occurs after the read or write.

One advantage of a read or write hardware breakpoint (also known as a data hardware breakpoint) is that you can set it on a specific memory location. When that location is accessed, the process will be halted, regardless of what instruction the process is executing.

- `mask=mask` — Specifies that the hardware breakpoint will be hit if the bitwise AND of the mask and the address about to be accessed is equal to the bitwise AND of the mask and the breakpoint address (`expr`). For example, you can use this feature to stop on accesses to every 16th byte in an array by providing a mask of `0x0000000F`. By default, `mask` is defined as `0xFFFFFFFF`.
- `data=data_value` — Specifies that the read/write hardware breakpoint will be hit if the value located at the given address and `data_value` (when masked with `dmask=data_mask`, if specified) are equal. For more information about the applicability of this option, see the description of the **Value** field in “Software and Hardware Breakpoint Editors” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in *MULTI: Debugging*.
- `dmask=data_mask` — Specifies that the read/write hardware breakpoint will be hit if the bitwise AND of the data mask and the value located at the given

address is equal to the bitwise AND of the data mask and the data value specified with `data=data_value`. For more information about the applicability of this option, see the description of the **Value Mask** field in “Software and Hardware Breakpoint Editors” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in *MULTI: Debugging*.

- `vm` — Instructs INTEGRITY to use a virtual memory breakpoint to simulate the hardware breakpoint. Because virtual memory breakpoints do not use any target-specific hardware breakpoint resources (INTEGRITY implements the mechanism instead), they are useful when the target lacks such resources. However, depending on access patterns, virtual memory breakpoints may significantly impact performance. The `vm` option is only supported if you are using a run-mode connection to debug INTEGRITY version 10 or 11 tasks in virtual AddressSpaces.
- `tm` and `live` — Specify a context—INTEGRITY TimeMachine or the live target—where the hardware breakpoint is active. If neither option is specified, the hardware breakpoint is active in both contexts.
- `--` — Indicates the end of the options applied to the hardware breakpoint, and signals that the following item is the address expression where the hardware breakpoint should be located. For example:

```
hardbrk write -- data
```

sets a hardware breakpoint on writes on the variable named `data`. This option is useful if the expression shares the name of a **hardbrk** option.

- `expr[:size]` — Specifies the location of the breakpoint and can be a memory address, variable, or pointer name. Specifying a size forces the breakpoint to cover an explicit number of bytes. Sizes available vary by target and may be limited to 1, 2, and 4 bytes or to power-of-2 bytes. A size of 1, 2, 4, or 8 bytes may also limit the memory accesses that hit the hardware breakpoint to accesses of the same length (for example, size 2 stops on a write of a `short`, but not a `char` or `int`). The default size is the size of the instruction for text locations, the size of the object for variables, and one byte for other memory locations.
- `{commands}` — Executes the list of `commands` each time the hardware breakpoint is hit. For information about setting such lists of commands, see “Using Command Lists in Debugger Commands” on page 12.

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

```
Stopped by hardware break on execute (main#12)
```

If the target system cannot support the requested breakpoint, an error message appears.

Example uses of the **hardbrk** command follow.

To delete hardware breakpoint number two:

```
> hardbrk delete=2
```

To delete all hardware breakpoints:

```
> hardbrk delete=*
```

To stop on any read from variable `val`:

```
> hardbrk read val
```

To stop on any read or write to locations `0x10000` to `0x1000f`:

```
> hardbrk read write mask=0xffffffff0 0x10000
```

To stop on any write to the first sixteen bytes pointed to by `string`:

```
> hardbrk write *string:16
```

To print accessed variable `val` in the command pane any time the variable `val` is accessed, and then resume the process:

```
> hardbrk val {echo "accessed variable val"; resume}
```

To stop when the instruction at address `0x100ff` is executed:

```
> hardbrk execute 0x100ff
```

**Note**

MULTI removes all hardware breakpoints when detaching from a process.

install_bp_on_request

install_bp_on_request -enable | -install

Controls when the Debugger installs breakpoints on the target. This command is intended for use in setup scripts in freeze-mode debugging, in situations where you have to run code on the target when it is not safe to install breakpoints on the target.

For example, this command may be applicable when:

- You are loading a program (such as an OS kernel) to your target which uses kernel virtual addresses that are different from physical addresses, and
- You have to run code at boot time on your target with the MMU disabled (such as a ROM monitor).

In such a scenario, in your setup script, you would first (before any code executes on your target) issue:

```
install_bp_on_request -enable
```

to prevent the Debugger from installing software breakpoints automatically. Then, when it is safe to install breakpoints (for example, when the target is stopped and the MMU is finally enabled), issue:

```
install_bp_on_request -install
```

to install all breakpoints and go back to normal breakpoint installation behavior.



Note

Most programs and targets do not require this command. We recommend that you contact Green Hills Support for help when you create your setup script.



Note

Restarting your program over a freeze-mode connection clears the `-enable` flag.



Note

While the contents of a monolith application may be shown in the target list under a freeze-mode connection before the kernel is booted and a

run-mode connection is established, any breakpoints set in monolith tasks only apply in a run-mode debugging environment. When the **install_bp_on_request** command is used in freeze mode (for example, in a setup script), it has no effect on breakpoints set in monolith tasks.

rominithbp

rominithbp -setup [location] | -finish | -remove

Sets or removes the post-initialization hardware breakpoint used in ROM and ROM-to-RAM copy debugging. This hardware breakpoint is used to signal to MULTI that ROM initialization is complete. For ROM-only programs, this breakpoint triggers the `-after rominit` Debugger hook (see “Hook Commands” on page 217). For ROM-to-RAM copy programs, this breakpoint also indicates that the program has been copied into RAM and that any ROM restrictions on debugging are no longer necessary. When the post-initialization hardware breakpoint is hit, MULTI performs all post-initialization actions, clears the breakpoint, and then resumes the target.

Available options are:

- `-setup [location]` — Sets the post-initialization hardware breakpoint at the specified location (if any). If you do not specify a location and if the application is ROM-run, MULTI sets the breakpoint at the first source line of `main()`. If the application is ROM-copy, MULTI sets the breakpoint at `__ghs_after_romcopy`, which defaults to one of the first instructions in RAM.
- `-finish` — Immediately performs all post-initialization actions (such as triggering the `-after rominit` Debugger hook, setting software breakpoints, etc.).
- `-remove` — Removes the post-initialization hardware breakpoint.

sb

sb *task action [%bp_label] [@bp_count] [&] [/off] [address_expression]*
[*{commands}*]

Sets target-specific breakpoints, where:

- *task* — Specifies which task(s) in the system the breakpoint is set on. The *task* argument is required and must be replaced by exactly one of the following letters:
 - *a* — Any task in the current address space.
 - *d* — Any attached task in the current address space.
 - *t* — The current task.
 - *u* — Any unattached task in the current address space.

Only the *a* and *t* options are supported with INTEGRITY.

- *action* — Specifies what action is taken when the breakpoint is hit. The *action* argument is required and must be replaced by exactly one of the following letters:
 - *n* — Notifies you.
 - *s* — Stops the system.
 - *t* — Stops the task that hit the breakpoint.

Only the *t* option is supported with INTEGRITY.

**Note**

The *task* and *action* arguments should not be separated by a space.

- *%bp_label* — Specifies a breakpoint label (see “Breakpoint IDs and Labels” on page 10).
- *@bp_count* — Specifies a breakpoint count (see “Debugger Command Conventions” on page 3).
- *&* — Causes the breakpoint to beep when it is hit.
- */off* — Inactivates the breakpoint. This allows you to set a breakpoint on a running target without halting the target.

- *address_expression* — Specifies an address (see “Using Address Expressions in Debugger Commands” on page 5).
- *{commands}* — Specifies a list of commands to be performed (see “Using Command Lists in Debugger Commands” on page 12).

Example uses of the **sb** command follow.

```
> sb tt foo#12
```

Sets a breakpoint at line 12 of the function `foo`. The breakpoint is triggered by the current task and stops the current task (this is equivalent to a standard breakpoint).

```
> sb at bar#12
```

Sets a breakpoint at line 12 of the function `bar`. The breakpoint is triggered by any task in the address space and stops the task that hit it.

setbrk

setbrk

GUI only

Sets a new breakpoint or removes an existing breakpoint at the current line (pointed to by the current line pointer) or at 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.

This command is very useful when bound to a mouse button (see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book).

As an alternative to using this command, you can click the breakdot to the left of the line to set or remove a breakpoint.

The **setbrk** command is different from the **tog** command, which you can use to toggle the status of an existing breakpoint (see “tog” on page 70).

The **setbrk 0** command has been replaced by “runtohere” on page 169.

sethbp

sethbp [enabled | disabled] [rolling] [{commands}]

Sets a hardware execute breakpoint at the current location in your program code. The hardware breakpoint will have the size of the instruction it is set on, an address at the current code location, and an access type of `execute`. Hardware breakpoints are implemented through direct hardware support and are only available on some targets. MULTI removes all hardware breakpoints when detaching from a process.

If no slots remain on the target for this hardware breakpoint, MULTI will first disable one of the existing hardware breakpoints that was set with the `rolling` attribute and then set the hardware breakpoint. If no rolling hardware breakpoints are set on the target and no slots are available, this hardware breakpoint will not be set.

The following optional arguments can be used with this command:

- `rolling` — Indicates that this hardware breakpoint may be disabled automatically to open a slot when you want to set a new hardware breakpoint in the future.
- `disabled` — Indicates that this hardware breakpoint is initially disabled and will consume no target resources.
- `enabled` — Indicates that this hardware breakpoint is initially enabled and will operate normally. If neither the disabled or enabled attribute is specified, this is the default.
- `{commands}` — Specifies one or more commands that will be executed when this hardware breakpoint is hit. See “Using Command Lists in Debugger Commands” on page 12.

stopif

stopif [/at | /tt] [file_relative_line_number] condition

Sets a conditional breakpoint at the line number specified. If no line number is specified, the current line number is used. (For more information, see “Specifying Line Numbers” on page 7 and “Using Address Expressions in Debugger Commands” on page 5.)

If you specify `/at`, the breakpoint is an any-task breakpoint. This is the default on targets that support any-task breakpoints. If you specify `/tt`, the breakpoint is task specific. This is the default on targets that do not support any-task breakpoints.

The *condition* must be an expression in the current language. When the process reaches the breakpoint, it will halt if *condition* evaluates to true; otherwise it will continue. 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, avoid expressions that begin with a number; otherwise the expression will be parsed incorrectly. For example, you should not do the following:

```
> stopif 5==y /*INCORRECT*/
```

In this example, the Debugger will try to set a breakpoint on line 5 with the condition `(==y)`, which will fail. To properly set this breakpoint, enclose the expression in parentheses, as follows.

```
> stopif (5==y) /*CORRECT*/
```

MULTI will do limited syntax checking to make sure that the *condition* is a valid expression.

See also “bif” on page 50.

stopifi

stopifi [`/at` | `/tt`] [*address*] *condition*

Sets a conditional breakpoint on the machine instruction at the specified address.

If you specify `/at`, the breakpoint is an any-task breakpoint. This is the default on targets that support any-task breakpoints. If you specify `/tt`, the breakpoint is task specific. This is the default on targets that do not support any-task breakpoints.

If *address* is not specified, the current machine instruction is used. The current machine instruction is set when the target halts at a new location or when you display a location using the `/i` or `/I` expression format. (For information about expression

formats, see “Expression Formats” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.)

If issued with an address, this command is identical to the **stopif** command, except that the breakpoint is placed on the instruction at the specified address rather than on the specified line (see “stopif” on page 68).

tog

tog [q] [on | off | tog] hbp [[at] *address_expression* | *hbp_id* | *]

tog [/force] [q] [on | off | tog] [b] [*address_expression* | *breakpoint_list* | *]

Toggles the active status of the specified breakpoint(s), where:

- q — Silent mode. Only error messages are printed.
- on — Activates all breakpoints matching the *address_expression* or *breakpoint_list*.
- off — Deactivates all breakpoints matching the *address_expression* or *breakpoint_list*.
- tog — Toggles the active status of all breakpoints matching the *address_expression* or *breakpoint_list*. This is the default if neither on nor off is specified.
- hbp — Toggles the status of a hardware breakpoint.
- at — Indicates that the hardware breakpoint specified by *address_expression* or *hbp_id* is an any-task hardware breakpoint.
- *address_expression* — An address expression that identifies the breakpoint to toggle. See “Using Address Expressions in Debugger Commands” on page 5.
- *hbp_id* — Numeric identifier of a hardware breakpoint.
- * — Use this to toggle all the hardware or software breakpoints (hardware if hbp is specified; software, otherwise).
- /force — Forces all breakpoints matching the *address_expression* to be toggled. If you do not specify this option and more than one breakpoint matches the *address_expression*, the **Select Breakpoints** dialog box appears, from which you can choose which breakpoints to toggle.

- `b` — Toggles the status of a software breakpoint. This is the default if `hbp` is not specified.
- `breakpoint_list` — A list of breakpoints to toggle. For more information, see “Breakpoint Ranges and Lists” on page 11.

Only existing breakpoints can be modified with this command. If there are none, an error message is displayed.

Tog

Tog [`q`] [`on` | `off` | `tog`]

Toggles the status of all software breakpoints. If no arguments are specified, `tog` is assumed.

- `q` — Silent mode. Only error messages are printed.
- `on` — Makes all software breakpoints active.
- `off` — Makes all software breakpoints inactive.
- `tog` — Toggles the active status of all software breakpoints.

This command is equivalent to:

tog /force [`q`] [`on` | `off` | `tog`] *

For more information, see “tog” on page 70.

Corresponds to: **Debug** → **Toggle All Breakpoints**

watchpoint

watchpoint *expr*

watchpoint -delete

The first format of the **watchpoint** command sets a watchpoint on the address indicated by *expr*, causing the process to halt when the address is written to.

The second format of the **watchpoint** command disables the software watchpoint.

When set, the watchpoint is implemented in one of the following ways:

- On systems that support hardware breakpoints, a hardware write breakpoint is set at the given address. If any-task hardware breakpoints are supported (as on INTEGRITY IoT and INTEGRITY version 10 and later), an any-task hardware write breakpoint is used. This type of breakpoint can be hit by any task in the address space in which it is set. See also “hardbrk” on page 59.
- If hardware breakpoints are not available, the watchpoint may be set using an efficient software hook. Only one watchpoint of this type may be set at a time. For information about enabling the software hook, see the documentation about run-time error checks in the *MULTI: Building Applications* book.

To disable this software watchpoint, enter:

```
watchpoint -delete
```

The `-delete` option only applies to the software watchpoint; it does not disable hardware watchpoints.

If neither of these methods is available, the watchpoint is not set.

Corresponds to: **Debug → Set Watchpoint**

Chapter 4

Building Command Reference

Contents

Building Commands	74
-------------------------	----

The commands in this chapter allow you to build a program, launch the MULTI Project Manager, or launch the **Utility Program Launcher**. For more information, see the *MULTI: Managing Projects and Configuring the IDE* book.

Building Commands

The following list provides a brief description of each building command. For a command's arguments and for more information, see the page referenced.

- **build** — Attempts to build the specified program (see “build” on page 74).
- **builder** — Opens the MULTI Project Manager (see “builder” on page 74).
- **wgutils** — Opens the **Utility Program Launcher**, which provides a GUI interface to Green Hills utility programs (see “wgutils” on page 75).

build

build [*program_or_project_name*]

GUI only

Attempts to build the given *program_or_project_name* located in the same project tree as the current executable. The build progress and error messages are displayed in a window. If no *program_or_project_name* is specified, the name of the current program is used.

Corresponds to: **Tools** → **Rebuild**

builder

builder

GUI only

Opens the MULTI Project Manager.

Corresponds to: **Tools** → **Project Manager**

wgutils

wgutils

GUI only

Opens the **Utility Program Launcher**, which provides a GUI interface to Green Hills utility programs. For more information, see the documentation about utility programs in the *MULTI: Building Applications* book.

Corresponds to: **Tools** → **Launch Utility Programs**

Chapter 5

Call Stack Command Reference

Contents

Call Stack Commands	78
---------------------------	----

The commands in this chapter allow you to view call stacks (also known as *call stack traces* or simply *stack traces*). A call stack lists the stack frames that are currently active in your program. Each stack frame typically represents a function call. Stack frames are shown in order from most to least recently created. For more information about viewing call stacks in the **Call Stack** window, see “Viewing Call Stacks” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

Call Stack Commands

The following list provides a brief description of each call stack command. For a command's arguments and for more information, see the page referenced.

- **calls** — Displays the current call stack (see “calls” on page 78).
- **callsview** — Displays the current call stack in the **Call Stack** window (see “callsview” on page 79).
- **cvconfig** — Configures the **Call Stack** window specified (see “cvconfig” on page 81).

calls

calls [*maxdepth*] [*par* | *nopar*] [*pos* | *nopos*] [*local* | *nolocal*] [*types* | *notypes*] [*templatetypes* | *notemplatetypes*] [*showallframes* | *noshowallframes*]

Displays the current call stack, where:

- *maxdepth* — Specifies the maximum number of stack frames you want to display. The default value of *maxdepth* is 20 and the maximum value is 32768.
- *par* | *nopar* — Specifies whether or not parameters passed to functions are displayed. If neither argument is specified, the parameters are displayed.
- *pos* | *nopos* — Specifies whether or not source positions of functions are displayed. If neither argument is specified, the source positions are displayed.
- *local* | *nolocal* — Specifies whether or not local variables used in functions are displayed. If neither argument is specified, the local variables are not displayed.

- `types | notypes` — Specifies whether or not the types of function parameters are displayed. This argument only affects functions within files compiled in C++ mode. If neither argument is specified, the types of function parameters are displayed.
- `templatetypes | notematetypes` — Specifies whether or not C++ template parameters are displayed or collapsed. This argument only affects functions within files compiled in C++ mode. If neither argument is specified, template parameters are displayed.
- `showallframes | noshowallframes` — Specifies whether or not the Debugger displays stack frames before `main()`. If neither argument is specified, stack frames before `main()` are not displayed.

You can view the call stack in its own window by using the **callsview** command (see “callsview” on page 79).

callsview

callsview [%name] [maxdepth] [par | nopar] [pos | nopos] [types | notypes] [templatetypes | notematetypes] [showallframes | noshowallframes] [win | nowin] [local | nocal]]

GUI only

Displays the current call stack in the **Call Stack** window, unless the `nowin` argument is specified (see below).

The optional arguments allow you to specify what information should be displayed, where:

- `%name` — Specifies a name for the window. `name` may be either a C string or an identifier in the style permitted by C (letters, numbers and underscores, beginning with a letter). `name` must not be the same as the name of any existing **Call Stack** window.
- `maxdepth` — Specifies the maximum number of stack frames you want to display. If this argument is not specified, the previously defined value is used. If no previously defined value exists, the default value is 20. The maximum value is 32768.

- `par | nopar` — Specifies whether the parameters passed to functions are displayed.
- `pos | nopos` — Specifies whether the source positions of functions are displayed.
- `types | notypes` — Specifies whether the types of function parameters are displayed. These arguments only affect functions within files compiled in C++ mode.
- `templatetypes | notematetypes` — Specifies whether template parameters are displayed or collapsed. These arguments only affect functions within files compiled in C++ mode.
- `showallframes | noshowallframes` — Specifies whether or not the Debugger displays stack frames before `main()`.
- `win | nowin` — Specifies where to display the call stack. If `win` is specified (the default), a **Call Stack** window is created. If `nowin` is specified, the call stack is printed to the command pane. `%name` should not be specified in conjunction with `nowin`.
- `local | nlocal` — Specifies whether or not the local variables used in functions are displayed. (This option is only applicable if `nowin` has been specified).

At the beginning of a debugging session, this command defaults to the following settings: `par pos types templatetypes noshowallframes win`. Subsequent calls to this command default to the previous configuration of the **Call Stack** window, except that `win` is always the default. Thus, if you change a setting other than `nowin` for the **Call Stack** window, it will be remembered the next time you open the **Call Stack** window (see “Viewing Call Stacks” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book).

Corresponds to: 

Corresponds to: **View** → **Call Stack**

cvconfig

cvconfig [%name] key [key]...

GUI only

Configures the **Call Stack** window identified by *name*. (The name of a **Call Stack** window is the same as the caption that appears in its title bar.) If no *name* is specified, **cvconfig** configures the last **Call Stack** window that was created or configured.

This command is primarily intended for use in scripts, since the functionality it provides is directly accessible from the GUI of the **Call Stack** window.

The *key* argument(s) can be directives with no arguments or parameters with assigned values. Acceptable values for *key* and their meanings are listed below:

- `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.
- `types` — Shows the types of function parameters.*
- `notypes` — Hides the types of function parameters.*
- `templatetypes` — Shows the template parameters.*
- `notemplatetypes` — Collapses template parameters into ellipses.*
- `showallframes` — Shows stack frames before `main()`.
- `noshowallframes` — Hides stack frames before `main()`.
- `edit` — Opens an Editor on the currently selected function.
- `local` — Displays local variables for the currently selected function in a Data Explorer.
- `copy` — Copies the contents of the **Call Stack** window to the clipboard.
- `print` — Prints the **Call Stack** window.

- `quit` — Closes the **Call Stack** window.
- `name=newname` — Renames the **Call Stack** window to *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.

*The parameter types and template parameter options only affect functions within files compiled in C++ mode.

Keys are handled sequentially in the order they are given. The keys `stop`, `refresh`, `help`, `copy`, and `quit` terminate the command when they are encountered, causing any remaining keys to be ignored. The same is true for any errors encountered during processing of the keys.

Keys and key values are case-insensitive.

For more information, see “Viewing Call Stacks” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

Chapter 6

Configuration Command Reference

Contents

General Configuration Commands	84
Button, Menu, and Mouse Commands	94

The configuration commands are broken up into two sections. The general configuration commands in the first section allow you to configure the Debugger or IDE. The button, menu, and mouse commands in the second section allow you to access or configure menus, toolbar buttons, key bindings, and mouse buttons.

For more information about configuring various aspects of the MULTI IDE, see Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

General Configuration Commands

The following list provides a brief description of each general configuration command. For a command's arguments and for more information, see the page referenced.

- **clearconfig** — Clears your default configuration for MULTI (see “clearconfig” on page 85).
- **configoptions** — Opens the **Options** dialog box (see “configoptions” on page 85).
- **configure** — Changes the value of a MULTI configuration option (see “configure” on page 86).
- **configurefile** — Reads the specified file, which must be a MULTI configuration file, and applies the options it describes to the current session (see “configurefile” on page 86).
- **fileextensions** — Performs operations that affect the file type and extension mappings used by the file choosers throughout MULTI (see “fileextensions” on page 87).
- **fontsize** — Increases or decreases the font size (see “fontsize” on page 87).
- **imagename** — Specifies that an executable will be running from the specified path, rather than from the source directory (see “imagename” on page 88).
- **loadconfigfromfile** — Opens a file chooser that allows you to select a MULTI configuration file to load into MULTI (see “loadconfigfromfile” on page 88).
- **saveconfig** — Saves the current configuration settings to the default configuration file (see “saveconfig” on page 88).

- **saveconfigtofile** — Saves the current configuration settings to a specified file (see “saveconfigtofile” on page 89).
- **setintegritydir** — Opens the **Default INTEGRITY Distribution** dialog box (see “setintegritydir” on page 89).
- **setuvelocitydir** — Opens the **Default u-velOSity Distribution** dialog box (see “setuvelocitydir” on page 90).
- **source** — Specifies directories for MULTI to search to find source files for the debugged executable (see “source” on page 90).
- **sourceroot** — Clears, lists, creates, or remaps the source root, which is the path MULTI prepends to each file's relative path (see “sourceroot” on page 91).
- **star** — Controls which entries are displayed in the target list (see “star” on page 92).
- **syncolor** — Sets syntax coloring options (see “syncolor” on page 93).

clearconfig

clearconfig

Clears your default configuration for MULTI. Note that this command affects the entire development environment.

Corresponds to: **Config** → **Clear User Default Configuration**

configoptions

configoptions

GUI only

Opens the **Options** dialog box.

Corresponds to: **Config** → **Options**

configure

configure *config_option* [= | :] *value*

configure *config_option*

configure ?

Changes the value of a MULTI configuration option. The `configure ?` command displays a list of configurable options. You can separate *config_option* from *value* with an equal sign (=), a colon (:), or a whitespace character (). If *value* is a Boolean, you can omit it and MULTI will toggle the option's setting.

An example use of this command, which changes MULTI's tab size to 9, follows:

```
> configure tabsize=9
```



Note

Do not use this command while the **Options** window is open.

For more information, see “Using the configure Command” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

configurefile

configurefile *file*

Reads *file*, which must be a MULTI configuration file, and applies the options it describes to the current session. MULTI configuration files can be created with the **saveconfigtofile** command (see “saveconfigtofile” on page 89).

For more information about configuration files, see “Creating and Editing Configuration Files” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

fileextensions

fileextensions *option filename*

GUI only

Performs operations that affect the file type and extension mappings used by the file choosers throughout MULTI. Each operation is performed on a file specified by *filename*. The behavior of this command depends on the value of *option*, which must be one of the following:

- `-load` — Loads in extension mappings from the file named *filename*. The extension mappings from that file replace all of the extension mappings throughout MULTI.
- `-save` — Creates a file named *filename* that can be loaded to reconstruct the extension mappings currently in use.
- `-append` — Inserts the extension mappings from the file named *filename* into the existing mappings. Existing extensions that are not modified by *filename* are preserved.

For more information, see “Configuring File Extensions” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

fontsize

fontsize `-inc` | `-dec`

Changes the size of the source code font, where `-inc` increments the size and `-dec` decrements the size. For more information about configuring fonts, see **Source Code Font** and **GUI Font** in “General Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

imagename

imagename [*path_to_executable*]

Specifies that an executable will be running from *path_to_executable*, rather than from the source directory.

By default, MULTI assumes that the final executable resides in the source directory where it gets built. If the executable will not be located in the source directory and/or the executable name will change, you must use this command to let MULTI know where the final executable will be running from. For example, some development tools automatically move the final executable out of the source directory and into another directory (sometimes renaming the executable in the process). The executable image in the alternative directory must be identical to the executable that is built in the source directory.

This command is only applicable if you are debugging an operating system such as Linux, Solaris, or Windows and the debug information for your program resides in a different location than that of the executable file.

loadconfigfromfile

loadconfigfromfile

GUI only

Opens a file chooser that allows you to select a MULTI configuration file to load into MULTI.

Corresponds to: **Config** → **Load Configuration**

saveconfig

saveconfig

Saves the current configuration settings to the default configuration file. MULTI reads this file upon startup to restore your configuration settings. See also “saveconfigtofile” on page 89.

For more information about configuration files, see “Creating and Editing Configuration Files” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

Corresponds to: **Config** → **Save Configuration as User Default**

saveconfigtofile

saveconfigtofile

GUI only

Saves the current configuration settings to a specified file.

This command is similar to the **saveconfig** command (see “saveconfig” on page 88), except that it opens a file chooser dialog box that allows you to choose a file in which to save the configuration. This can be useful when used with the command **configurefile** (see “configurefile” on page 86).

For more information about configuration files, see “Creating and Editing Configuration Files” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

Corresponds to: **Config** → **Save Configuration As**

setintegritydir

setintegritydir

GUI only

Opens the **Default INTEGRITY Distribution** dialog box in which you can provide MULTI with the location of the installed INTEGRITY distribution. This information is used to add INTEGRITY documentation to MULTI's **Help** menu and to determine the default INTEGRITY distribution (used by the **Project Wizard**). For more information, see “Configuring MULTI for Use with INTEGRITY or u-velOSity” in Chapter 2, “MULTI Tutorial” in the *MULTI: Getting Started* book.

Corresponds to: **Config** → **Set INTEGRITY Distribution**

setuvelocitydir

setuvelocitydir

GUI only

Opens the **Default u-velOSity Distribution** dialog box in which you can provide MULTI with the location of the installed u-velOSity distribution. This information is used to add u-velOSity documentation to MULTI's **Help** menu and to determine the default u-velOSity distribution (used by the **Project Wizard**). For more information, see “Configuring MULTI for Use with INTEGRITY or u-velOSity” in Chapter 2, “MULTI Tutorial” in the *MULTI: Getting Started* book.

Corresponds to: **Config → Set u-velOSity Distribution**

source

source [*num*] [*dir*]...

source - [*dir*]...

Specifies directories for MULTI to search to find source files for the debugged executable.

If, in the first format of the **source** command, *num* is specified, the directory numbered *num* in the current source path is replaced by the new one given by *dir*. If *num* is specified without a replacement directory, the specified entry is deleted from the list. The directory list is zero-based. If *num* is not specified, the specified directories are added to the list. If no arguments are specified, this command lists the directories that will be searched.

The second format of the **source** command discards any previously specified directories, replacing them with *dir* (if provided).

In either command format, you can specify multiple directories by separating them with spaces. If *dir* itself contains a space, enclose *dir* in quotation marks. On Linux/Solaris, directory names may include a tilde (~) as an abbreviation for the home directory.

**Note**

If MULTI cannot find the debugging information files associated with an executable, you must use the **loadsym** command (see “loadsym” on page 27) to specify the location of the files.

Corresponds to: **View** → **Source Path**

sourceroot

sourceroot [global] [clear | list | *new_root* | remap *old_dir1* [,*old_dir2*]... *new_dir1* [,*new_dir2*]...]

Clears, lists, creates, or remaps the source root, which is the path MULTI prepends to each file's relative path, where:

- *global* — Applies source root changes to all contexts—stand-alone executables and AddressSpaces—in the debugging session. Otherwise, source root changes are limited to the stand-alone executable or AddressSpace from which the command is executed.
- *clear* — Clears the current source root.
- *list* — Lists the current source root.
- *new_root* — Creates a new root that is prepended to the relative paths of files.
- *remap old_dir1* [,*old_dir2*]... *new_dir1* [,*new_dir2*]... — Sets one or more remappings, removing any previously existing remappings and source roots. For example, if two remappings are specified, *old_dir1* remaps to *new_dir1*, and *old_dir2* remaps to *new_dir2*.

Both the old and new directories are case-sensitive. If you specify multiple remappings, use a comma to separate the directories; there should not be any spaces between *old_dir1* and *old_dir2* or between *new_dir1* and *new_dir2*. If a path contains a space, enclose the path in quotation marks.

Defining a new source root allows you to build an executable with source files from one location and debug that same executable with those source files accessible in a different location.

For example, if you build a project using source files from **C:\checkout\src**, MULTI locates the executable and its supporting files relative to this location. Now suppose

you want to debug on a computer where the source files are accessible in **X:\src**. You can change the source root by specifying:

```
> sourceroot remap C:\checkout\src X:\src
```

You can specify multiple source locations. For example, if you build a project using source files from **/usr/local/proj1/src** and **/usr/local/shared/src**, and you want to debug on a computer where the source files are accessible in **/machine2/src1** and **/machine2/srcshared**, you can change both source roots with one command by specifying:

```
> sourceroot remap /usr/local/proj1/src,/usr/local/shared/src \
    /machine2/src1,/machine2/srcshared
```

For more information, see the documentation about building and debugging on different hosts in the *MULTI: Building Applications* book.

star

star showstarred | showall | toggle

star set | clear "target_list_entry" ["target_list_sub_entry"]...

The first format of the **star** command shows only starred entries in the target list, or it shows all entries.

The second format of the **star** command stars/unstars target list entries.

This command takes the following arguments:

- **showstarred** — Shows only starred target list entries, their ancestors (if any), the currently selected target list entry, and stopped entries.
- **showall** — Shows all target list entries.
- **toggle** — Toggles between the behavior of **showstarred** and **showall**.
- **set** — Stars the specified target list entry.
- **clear** — Unstars the specified target list entry.

- `"target_list_entry" ["target_list_sub_entry"]...` — Specifies which target list entry should be starred or unstarred. The string must match the entry in the target list exactly. For example:

```
star set task2
```

stars `task2` in the target list. If there are multiple `task2` target list entries, all are starred. To star only one of them, you can specify its unique parent, unique grandparent + parent, etc. For example:

```
star set myproject AddressSpace1 task2
```

stars `task2` in `AddressSpace1` in `myproject`. `AddressSpace1` and `myproject` are not starred.

When specifying the ancestor(s) of the target list entry you want to star/unstar, you may not skip levels. For example, the following is not permitted:

```
star set myproject task2
```

because `task2` is not a child of `myproject`; it is a child of `AddressSpace1`. However, the following *is* permitted:

```
star set AddressSpace1 task2
```

For more information, see “The Star Column” in Chapter 2, “The Main Debugger Window” in *MULTI: Debugging*.

syncolor

syncolor [0 | 1] [a] [C] [k] [d] [n] [s] [c]

GUI only

Sets syntax coloring options.

- 0 (zero) — Turns off syntax coloring for all categories.
- 1 (one) — Turns on syntax coloring for all categories.
- a — Toggles syntax coloring for all categories.

- `c` (lowercase) — Toggles syntax coloring for character constants.
- `C` (uppercase) — Toggles syntax coloring for comments.
- `d` — Toggles syntax coloring for dead code.
- `k` — Toggles syntax coloring for language keywords.
- `n` — Toggles syntax coloring for numbers.
- `s` — Toggles syntax coloring for string constants.

For example, the command `syncolor 0Ck` turns off syntax coloring for all categories, then toggles it on for comments and language keywords, while `syncolor 1d` turns on syntax coloring for all categories, then toggles it off for dead code. Without any arguments, **syncolor** prints the present state of all the categories.

Button, Menu, and Mouse Commands

The following list provides a brief description of each button, menu, and mouse command. For a command's arguments and for more information, see the page referenced.

- `->` — Opens the specified menu (see “->” on page 95).
- **customizemenus** — Opens the **Customize Menus** window, which allows you to create new menus or edit existing menus for a number of MULTI tools (see “customizemenus” on page 95).
- **customizetoolbar** — Opens the **Customize Toolbar** window, which allows you to rearrange the order of buttons on the Debugger toolbar, add pre-defined and custom buttons, and delete buttons (see “customizetoolbar” on page 95).
- **debugbutton**, **editbutton** — Adds, deletes, or configures a button on the Debugger's or Editor's toolbar (see “debugbutton, editbutton” on page 96).
- **inspect** — Opens a shortcut menu on the specified string (see “inspect” on page 98).
- **keybind** — Binds a key to a command (see “keybind” on page 98).
- **menu** — Defines a menu item to attach to a menu bar, MULTI button, mouse button, or keyboard key (see “menu” on page 99).
- **mouse** — Defines the function of the mouse buttons (see “mouse” on page 99).

->

-> *menu_name*

GUI only

Opens the menu *menu_name*. For example:

>-> FileMenu

opens the **File** menu.

This command can also be used to create submenus. For more information, see “Configuring and Customizing Menus” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

customizemenus

customizemenus

GUI only

Opens the **Customize Menus** window, which allows you to create new menus or edit existing menus for a number of MULTI tools. For more information, see “Configuring and Customizing Menus” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

Corresponds to: **Config** → **Customize Menus**

customizetoolbar

customizetoolbar

GUI only

Opens the **Customize Toolbar** window, which allows you to rearrange the order of buttons on the Debugger toolbar, add pre-defined and custom buttons, and delete buttons. For more information, see “Adding, Removing, and Rearranging Toolbar Buttons” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book.

Corresponds to: **Config** → **Customize Toolbar**

debugbutton, editbutton

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

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

GUI only

Adds, deletes, or configures a button on the Debugger's (**debugbutton**) or Editor's (**editbutton**) toolbar, where:

- *num* is the number that MULTI assigns to the button.
- *name* is the name of the button. You must enclose the name in quotation marks if it contains any character that is not a letter, digit, or underscore.
- *command* is the command executed when the button is pressed. You may use semicolons to execute multiple commands. For example:

```
> debugbutton printxy c="print x;print y"
```

This command creates a button named **printxy** that, when clicked, prints out the values of the variables *x* and *y* in the current context.

- *iconname* is the name of the button's icon, which may be:
 - A built-in icon. To obtain the names of built-in icons and to see what the icons look like, select **Config** → **Options** → **MULTI Editor** → **Configure Editor Buttons** or, from the Debugger, select **Config** → **Customize Toolbar** → **Add Custom Button** ().
 - The filename of an icon you have created yourself. If only a partial path is given, it is assumed to be relative to the MULTI IDE installation directory. For information about creating icons, see “Creating and Working with Icons” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.
- *helpstring* is the text that appears in the status bar when the cursor moves over the button.
- *tooltip* is the text that appears when the cursor hovers over the button. If you do not specify a tooltip, the name of the button is used.

The arguments *command*, *iconname*, *helpstring*, and *tooltip* must be entered as either single words or quoted strings of the form:

"This is a quoted string."

"These are quotes \" \" within a quoted string."

These commands have the following several special forms:

- **debugbutton** or **editbutton** — Lists defined Debugger or Editor buttons. The **debugbutton** command does not list the **Close Debugger** button or the separator before it; these cannot be modified or deleted.
- **debugbutton 0** or **editbutton 0** — Deletes all Debugger or Editor buttons, with the exception that the **debugbutton 0** command does not delete the **Close Debugger** button or the separator before it.
- **debugbutton num** or **editbutton num** — Deletes the Debugger or Editor button numbered *num*. You can view button numbers by entering **debugbutton** or **editbutton** in the command pane.
- **debugbutton -** or **editbutton -** — Adds a separator to the end of the toolbar. The separator is only visible if a button is added after it. Separators cannot be deleted by name.
- **debugbutton name [...]** or **editbutton name [...]** — If no optional arguments are specified, deletes the button named *name*. If optional arguments are specified and if a button named *name* exists, the button is replaced. Otherwise a new button named *name* is added to the end of the Debugger or Editor toolbar. You can view button names by entering **debugbutton** or **editbutton** in the command pane.
- **debugbutton num name [...]** or **editbutton num name [...]** — Replaces the button numbered *num* with a new button named *name*. You can view button numbers by entering **debugbutton** or **editbutton** in the command pane.

You cannot save **debugbutton** changes across MULTI sessions. As a result, this command is generally only useful for the creation or modification of buttons executed by scripts. For information about how to change the Debugger toolbar permanently, see “Adding, Removing, and Rearranging Toolbar Buttons” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book.

To save **editbutton** changes across MULTI sessions, select **Config → Save Configuration as User Default**.



Note

The **debugbutton** command does not affect customizations you make through the **Customize Toolbar** window. For information about this window, see “Adding, Removing, and Rearranging Toolbar Buttons” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book.

inspect

inspect [*string*]

GUI only

Opens a shortcut menu on *string*, equivalent to the default behavior when you right-click *string*.

This command is generally bound to a mouse click using the **mouse** command (see “Customizing Mouse Behavior with the mouse Command” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book).

See also “`browse`ref, `xref`” on page 295.

keybind

keybind [*location*]

keybind *key* [*modifiers*] [*@location*] [=*command*]

GUI only

Binds a key to a command. For a complete description of this command, see “Customizing Keys with the keybind Command” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

menu

menu [*name*] [{ { *label* [*'label'*] *cmd* } }]

GUI only

Defines a menu item to attach to a menu bar, MULTI button, mouse button, or keyboard key. For a complete description of this command, see “Configuring and Customizing Menus” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

The **l m** (lowercase **l**, uppercase **m**) command lists the menus defined with this command (see “l” on page 115).

mouse

mouse [*location*]

mouse *button_num* *Click*click_num* [(AtOnce)] [|*modifiers*] @*location* =*command*

GUI only

Defines the function of the mouse buttons. For a complete description of this command, see the “Customizing Mouse Behavior with the mouse Command” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.



Note

The command arguments should not be separated by spaces.

Chapter 7

Debugger Note Command Reference

Contents

Debugger Note Commands	102
------------------------------	-----

The commands in this chapter allow you to create, access, and delete Debugger Notes. Debugger Notes allow you to associate notes with any line of source or assembly code. For more information about Debugger Notes, see Chapter 10, “Using Debugger Notes” in the *MULTI: Debugging* book.

Debugger Note Commands

The following list provides a brief description of each Debugger Note command. For a command's arguments and for more information, see the page referenced.

- **notedel** — Deletes one or more Debugger Notes (see “notedel” on page 102).
- **noteedit** — Creates a new Debugger Note with the properties specified, or modifies an existing Note (see “noteedit” on page 103).
- **notelist** — Lists all Debugger Notes that exist for the current executable (see “notelist” on page 103).
- **notestate** — Loads Debugger Notes from the specified file, or saves the current executable's list of Debugger Notes to the specified file (see “notestate” on page 104).
- **noteview** — Navigates to the location of the specified Debugger Note or opens a Note Browser displaying all Debugger Notes for the program being debugged (see “noteview” on page 104).

notedel

notedel [*@num* | *address_expression* | -all]

Deletes a Debugger Note with the number specified by *@num* or at the location specified by *address_expression*. (For information about using an *address_expression* to specify a location, see “Using Address Expressions in Debugger Commands” on page 5.) If -all is specified, this command deletes all Debugger Notes. If no argument is specified, the Note at the current position is deleted. If the last Note in a group is deleted, that group is also deleted.

noteedit

noteedit [*@num*] [*address_expression*] [-name *name*] [-group *group_name*] [-text *text* | -prepend *text* | -append *text*] [-noedit]

Creates a new Debugger Note with the properties specified, or modifies an existing Note.

- *@num* — Specifies the number of the Debugger Note that will be modified.
- *address_expression* — Specifies the location of the Debugger Note that will be modified. See “Using Address Expressions in Debugger Commands” on page 5.
- -name *name* — Specifies the name of the Debugger Note.
- -group *group_name* — Specifies the group to which the Debugger Note belongs.
- -text *text* — Specifies the text contained in the Debugger Note.
- -prepend *text* — Prepends *text* to an existing Debugger Note.
- -append *text* — Appends *text* to an existing Debugger Note.
- -noedit — Suppresses the **Edit Note** dialog box.

If *@num* is specified, or *@num* is not specified and the location specified already contains a Note, that Note will be edited. If *name*, *group*, or *text* is specified, it will be changed in the Note.

notelist

notelist [-menu]

Lists all Debugger Notes that exist for the current executable. If no argument is specified, the Notes are listed in the command pane.

If -menu is specified, a shortcut menu displays the names of the Notes, which you can use to navigate to one of the Notes. The Notes are divided by group with the group name dimmed at the beginning of each group. In addition, the four most recently used Notes are displayed at the top of the menu.

notestate

notestate [-load *filename* | -save *filename*]

Loads Debugger Notes from the specified file, or saves the current executable's list of Debugger Notes to the specified file.

If you load Notes from a file, they are merged into the executable's existing list of Notes (if any). Notes that already exist at a specific location are not replaced by Notes loaded from the file.

noteview

noteview [@num | -prev | -next | -error]

Navigates to the location of the specified Debugger Note or opens a Note Browser displaying all Debugger Notes for the program being debugged, where the arguments have the following effects:

- @num — Goes to the specified Note.
- -prev — Goes to the previous Note.
- -next — Goes to the next Note.
- -error — Displays the error string from the last time a Debugger Note state was loaded.

If no argument is specified, the **Note Browser** window opens.

Corresponds to: **View** → **Debugger Notes**

Chapter 8

Display and Print Command Reference

Contents

Display and Print Commands	106
----------------------------------	-----

The commands in this chapter allow you to print information to the command pane and control certain aspects of the MULTI Debugger's display. For information about the main Debugger window's display, see Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

Display and Print Commands

The following list provides a brief description of each display and print command. For a command's arguments and for more information, see the page referenced.

- **asmmode** — Changes target-specific mode variables (see “asmmode” on page 108).
- **assem** — Controls the display mode of the Debugger window's source pane (see “assem” on page 108).
- **cat** — Prints the contents of the specified files to the command pane (see “cat” on page 109).
- **clear** — Clears the command pane, target pane, I/O pane, serial terminal pane, Python pane, or traffic pane of the Debugger (see “clear” on page 109).
- **comeback** — Restores the Debugger and debug-related windows after the **goaway** command has hidden them (see “comeback” on page 109).
- **components** — Lists components and their aliases (see “components” on page 110).
- **dbprint** — Prints the source currently being viewed in the Debugger (see “dbprint” on page 110).
- **debugpane** — Changes the Debugger command pane to the specified pane (see “debugpane” on page 111).
- **dumpfile** — Writes the contents of the Debugger's source pane or the file on display in the Debugger to a text file (see “dumpfile” on page 111).
- **E** — Displays the source code corresponding to the stack frame you specify (see “E” on page 112).
- **echo** — Prints the given text to the command pane, removing quotation marks if there are any (see “echo” on page 113).
- **eval** — Evaluates the given expression (see “eval” on page 113).
- **examine** — Examines the given object (see “examine” on page 114).

- **goaway** — Hides the Debugger and debug-related windows (see “goaway” on page 114).
- **l** — Lists various Debugger objects (see “l” on page 115).
- **map** — Lists the section address map for the current program, including section starts, ends, and sizes (see “map” on page 117).
- **mprintf** — Prints to the command pane (see “mprintf” on page 117).
- **mrulist** — Allows you to display and modify the contents of the most recently used lists (see “mrulist” on page 118).
- **mute** — Controls how much output from Debugger commands is displayed (see “mute” on page 119).
- **p, print** — Displays the value of the given expression using the specified format (see “p, print” on page 119).
- **printline** — Prints the specified number of source lines, starting at the specified line number (see “printline” on page 119).
- **printphys** — Takes in a numerical address or an address expression and prints the physical address mapped to that virtual address (see “printphys” on page 120).
- **printwindow** — Prints a specified section of source code (see “printwindow” on page 120).
- **pwd** — Prints MULTI's current working directory (see “pwd” on page 120).
- **Q** — Sets or toggles the Debugger's quiet mode (see “Q” on page 121).
- **savedebugpane** — Saves the contents of the specified Debugger pane to the specified file (see “savedebugpane” on page 121).
- **windowcopy** — Copies the specified window's current selection to the clipboard (see “windowcopy” on page 122).
- **windowpaste, windowspaste** — Pastes the current selection into the input buffer of the specified window (see “windowpaste, windowspaste” on page 122).

asmmode

asmmode -reset | -query [*name*] | -set *name*[=*val*] | -clear *name* | -toggle *name*

Changes target-specific mode variables, which may affect the behavior of the disassembler and/or other target-specific behavior in the Debugger.

Whether this command is available, and which variables it allows you to manipulate, depends on the Compiler release you are using and the target architecture of the program you have selected in the Debugger.

Available arguments are:

- **-reset** — Sets all mode variables to zero.
- **-query** [*name*] — If no name is specified, prints a list of all mode variables and their respective values. Names you can use to manipulate the variables are shown in parentheses.

If a name is specified, prints only the value of the named variable.

- **-set** *name*[=*val*] — Sets mode variable *name* to the specified value *val*, or to 1 if no value is specified.
- **-clear** *name* — Sets mode variable *name* to zero.
- **-toggle** *name* — If the mode variable *name* is nonzero, sets it to zero. If it is zero, sets it to 1.

If you need more information about the semantics of a specific mode variable, please contact Green Hills support.

assem

assem [on | off | tog | nosource]

GUI only

Controls the display mode of the Debugger window's source pane, where:

- **on** — Interlaces the appropriate disassembled instructions between lines of source code. This option corresponds to **View** → **Display Mode** → **Interlaced Assembly** and the  button when selected.

- `off` — Shows source code only. This option corresponds to **View → Display Mode → Source Only** and the  button when not selected.
- `tog` — Toggles the display between source code only and interlaced source code with disassembly. This is the default.
- `nosource` — Shows disassembly only. This option corresponds to **View → Display Mode → Assembly Only**.

If this option is used and the current file contains no program code (such as a header file), the source file is displayed, but if you navigate to another file that contains program code, the Debugger displays disassembly only.

cat

cat *filename* [*filename*]...

Prints the contents of the specified files to the command pane. Multiple files will be printed in the order specified, one after the other. If you specify a filename containing a space, you must enclose *filename* in double quotation marks.

clear

clear [`cmd` | `target` | `io` | `serial` | `python` | `traffic`]

GUI only

Clears the command pane, target pane, I/O pane, serial terminal pane, Python pane, or traffic pane of the Debugger. With no arguments, this command clears the pane that is currently visible in the Debugger. With `cmd`, `target`, `io`, `serial`, `python`, or `traffic` specified, this command clears the specified pane.

comeback

comeback

GUI only

Restores the Debugger and debug-related windows after the **goaway** command has hidden them (see “goaway” on page 114).

The **goaway** and **comeback** commands are only useful when MULTI is being externally controlled via a command script because there is no way to interactively issue **comeback** after **goaway** has hidden the Debugger.

components

components [*component_name*]

Lists components and their aliases. When run without the *component_name* argument, **components** lists the unique name of all components in the system along with a short list of aliases (if any) for each component. When a specific component name is specified, **components** lists that component's unique name along with all of its aliases.

Unique names are of the form:

component.number

where *number* is unique for each component, and increases from 1. The unique names of components may change between releases as new types of components are added.

To create a new alias for a new component, use the **new** command (see “new” on page 31).

To route commands to components, use the **route** command (see “route” on page 201).

dbprint

dbprint [*w* | *f*]

GUI only

Prints the source currently being viewed in the Debugger. If *f* is specified, the entire source file will be printed. If *w* is specified, only the source lines that are currently visible in the Debugger window will be printed. If you run this command without arguments, it has the same effect as `dbprint w`.

The **dbprint** *w* command corresponds to **File** → **Print Window**.

The **dbprint** ε command corresponds to **File** $\rightarrow$ **Print**.

debugpane

debugpane [cmd | target | io | serial | python | traffic | next | prev]

GUI only

Changes the Debugger command pane to the specified pane. There are six possible panes: the command pane, the target pane, the I/O pane, the serial terminal pane, the Python pane, and the traffic pane.

Specifying `cmd`, `target`, `io`, `serial`, `python`, or `traffic` will switch to that pane. Specifying `next` or `prev` will switch to the next or previous pane (in the order: command pane, target pane, I/O pane, serial terminal pane, Python pane, traffic pane). With no arguments, `debugpane` will switch to the next pane. See also “savedebugpane” on page 121.

In addition to this command, the Debugger also includes six tabs that allow you to switch panes. See “The Cmd, Trg, I/O, Srl, Py, and Tfc Panes” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

dumpfile

dumpfile [filename]

GUI only

Writes the contents of the Debugger's source pane to a text file, or, if debug information is available, writes the entire file. You can use this command to capture your program's disassembly, as well as source interlaced with your program's disassembly. The output is written to the specified file, creating the file if necessary. If no file is specified, you are prompted to name one.

E

E [*stack* | +[*num*] | -[*num*]]

Displays the source code corresponding to the stack frame you specify. If there is no source for the instructions near the program counter, the Debugger will display the disassembly of your program at that location.

The **E** command can be used in the following forms:

- **E** — Displays the function at the top of the stack. This option corresponds to **e 0_** (see “e” on page 147), , and **View → Navigation → Current PC**.
- **E *stack*** — Displays the function at call stack frame number *stack*. Equivalent to the **e *num_*** command (see “e” on page 147).
- **E +[*num*]** — Displays the function *num* functions above the currently visible function on your process's call stack, where *num* defaults to 1 if not specified. For example, **E +1** displays one function above the current one on the stack. This is different from **E 1**, which displays the function at stack frame number one.

This option corresponds to  and **View → Navigation → UpStack**.

- **E -[*num*]** — Displays the function *num* functions below the currently visible function on your process's call stack, where *num* defaults to 1 if not specified. For example, **E -1** displays one function below the current one on the stack. This is different from **E 1**, which displays the function at stack frame number one.

This option corresponds to  and **View → Navigation → DownStack**.

echo

echo *text*

Prints the given *text* to the command pane, removing quotation marks if there are any. For example, both of the following give the same result:

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

This command differs from the **print** command, in that the Debugger does not attempt to evaluate the given text as a programmatic expression (see “p, print” on page 119).

eval

eval *expr*

Evaluates *expr*, which is an expression in the current source language. Note that this command may read from and write to target memory. This command is similar to the **print** command (see “p, print” on page 119), except that it does not echo the results. This command should be used instead of the **print** command when performing I/O accesses, since printing the result of *expr* may cause an extra read of the I/O address.

For example:

```
> eval *(int *)0xfffffa0c0 = 0x123
```

will write 0x123 to the address 0xfffffa0c0. For information about accessing I/O memory, see also the system variable `_CACHE` in “System Variables” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

examine

examine [/format] *expr*

examine *address_expression*

examine *numberb*

Examines the given object, possibly with the given formatting options. This command has three forms:

- When the given argument is [/format] *expr*, **examine** is equivalent to **print** /format *expr*, which evaluates the expression and prints the result with the given formatting options (see “p, print” on page 119).
- When the given argument is *address_expression*, **examine** is equivalent to **e** *address_expression*; **p** *address_expression*, which displays the given location in the source pane and then prints the address in the command pane. See “e” on page 147 and “p, print” on page 119.
- When the given argument is *numberb*, **examine** causes MULTI to display the location where the breakpoint with the ID *number* is set in the Debugger source pane.

goaway

goaway

GUI only

Hides the Debugger and debug-related windows such as the Data Explorer, the **Register View** window, and the **Memory View** window. MULTI Editor windows that were opened from the Debugger are also hidden. Use the **comeback** command to restore these windows (see “comeback” on page 109).

The **goaway** and **comeback** commands are only useful when MULTI is being externally controlled via a command script because there is no way to interactively issue **comeback** after **goaway** has hidden the Debugger.

l

l [*object* [*string*]]

l T | **l Tx** [[-addressSpace *address_space_name*] -taskName *task_name* | -taskId *task_id*]

l ?

(This command is a lowercase **L**.)

Lists various Debugger objects. In the first format of this command, the *string* argument typically restricts file lists to objects whose names contain *string* and other lists to objects whose names start with *string*. Exceptions are stated below.

The second format of this command lists run-mode tasks. **l T** prints at most 15 characters per task name. **l Tx** does not limit the number of characters printed.

The third format of this command prints the command usage.

Permitted values for *object* are as follows:

- No parameter — Lists local variables and parameters of the current function, which must be on the stack. If the current function is a C++ instance method, this lists the `this` pointer as well.
- @ — Lists the addresses of local variables. For this type of object, if *string* is specified, it is interpreted as a function name, and variables local to that function are listed. The function must be on the stack.
- b — Lists breakpoints (*string* parameter ignored). Identical to the **B** command (see “B” on page 48).
- c — Lists all types, including structs and classes.
- d — Lists directories that will be searched for source files (*string* parameter ignored). In the list, directories that can be manipulated using the **source** command are prefixed with numbers. See “source” on page 90.
- D — Lists all dialog boxes (*string* parameter ignored).
- f — Lists the names of all source files.
- F — Lists functions and their addresses. If the optional *string* parameter is given and it contains a dot (.), only functions in file *string* are displayed. If

the *string* parameter is given and it does not contain a dot, it is treated as a pattern that may contain wildcards, and only functions matching the pattern are displayed. An asterisk (*) in the list indicates that the function has no debugging information.

- **g** — Lists the names and addresses of all global variables, as well as static variables that are in scope for the current file.
- **i** — Lists source files included by the currently displayed file (*string* parameter ignored).
- **m** — Lists functions with their mangled names. Same as **l f** except that the mangled names of C++ functions are also listed.
- **M** — Lists menus defined with the **menu** command. If the optional *string* parameter is provided, lists the contents of the menu whose name has been specified. See “Configuring and Customizing Menus” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.
- **P** — Lists processes (*string* parameter ignored).
- **r** — Lists registers. See also “The View Menu” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book and “regview” on page 195.
- **R** — Lists register synonyms.
- **s** — Lists MULTI special variables. System variables that begin with an underscore (_) represent the internal state of the Debugger and are excluded by default, but you can list them with **l s _**.
- **S** — Lists static variables. If the optional *string* parameter is given and it contains a dot (.), only static variables in file *string* are displayed. If the *string* parameter is given and it does not contain a dot, only static variables whose names start with *string* are displayed.
- **t** — Lists type definitions.
- **z** — Lists signals (*string* parameter ignored).
- **func** — Lists all locals and parameters of the function *func* (which must be on the stack). If *func* is a C++ instance method, this argument lists the `this` pointer as well. If *func* starts with an @, the addresses of all locals, parameters, and `this` are printed.

Permitted options to `l T` and `l Tx` are:

- `-addressSpace address_space_name` — Restricts the list of tasks to those in the address space named `address_space_name`.
- `-taskName task_name` — Restricts the list of tasks to those named `task_name`.
- `-taskId task_id` — Restricts the list to the task with ID `task_id`.

Corresponds to: **View** → **List** → *Object*

map

map [*filename* | `-modules` | `-find address`]

Lists the section address map for the current program, including section starts, ends, and sizes.

- *filename* — Lists the section address map for the specified module only.
- `-modules` — Lists the names and locations of the currently loaded modules.
- `-find address` — Lists the module and section that contains the specified memory address.

mprintf

mprintf (*format_string*, ...)

Prints to the command pane. This command takes the same syntax as the C library `printf()` function, except that the `%n` conversion specifier is not supported.

For example, given the following target code:

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

And with the following command:

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

The Debugger will display:

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

mrulist

mrulist *subcommand* [*args*]

GUI only

Allows you to display and modify the contents of the most recently used (MRU) lists. Two examples of MRU lists are the recent files list (with listname `EditedFiles`) and the recent projects list (with listname `Projects`).

The argument *subcommand* is required and must be one of the following:

- `listall` — Lists the name of each existing MRU list.
- `print listname` — Lists the contents of the specified MRU list.
- `insert listname slot entry` — Inserts an entry into the specified MRU list at the specified slot number. The entry will be placed in the specified slot, and all previous entries at or below the slot will be shifted down by one. If the slot number is greater than the number of entries, the entry will be added to the end of the list. The slot numbers are zero based, and the maximum number of entries is nine; therefore the useful range of slot numbers is 0 to 8.
- `delete listname [slot]` — Deletes an entry or all entries from the specified MRU list. If a slot is specified, the entry at the slot is deleted. If no slot is specified, all entries are deleted.
- `change listname slot entry` — Changes the entry in the specified MRU list at the specified slot.
- `update listname entry` — Moves the specified entry to the first slot of the MRU list if the entry already exists in the list, or inserts the entry at the top of the list if the entry does not already exist in the list.

mute

mute *state*

Controls how much output from Debugger commands is displayed. The argument *state* is required and must be one of the following:

- `off` — All output from Debugger commands is displayed. This is the default setting.
- `some` — Only serious error messages are displayed. All other output is suppressed.
- `on` — All output from Debugger commands is suppressed, including all error messages.

p, print

p [/format] *expr*

print [/format] *expr*

Displays the value of the expression *expr*, using the format *format*, by evaluating the expression exactly as the current language does. The expression *expr* can be any expression in the current language. For a list of available formats, see “Expression Formats” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

See also “echo” on page 113, “eval” on page 113, and “examine” on page 114.

Corresponds to: **View** → **Print Expression**

printline

printline [*count* [*line*]]

Prints *count* source lines, starting at the file-relative line number *line*. If *count* is not specified, one line is printed. If *line* is not specified, 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 in GUI mode.

In non-GUI mode, entering a line number prints out that line.

printphys

printphys [*address* | *expression*]

Takes in a numerical address or an address expression and prints the physical address mapped to that virtual address. The following example output is from a Power Architecture INTEGRITY target being debugged in freeze mode:

```
> printphys $pc
Virtual:    0x00010318 (main)
Physical:   0x00345318
```



Note

This command is not supported in run mode or if a TimeMachine-enabled task is selected in the target list and the target processor does not support data trace.

printwindow

printwindow [*line* [*num*]]

Prints a section of source code, *num* lines long, centered around the file-relative line *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” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book. The current line is indicated by a greater-than sign (>) between the line numbers and the source code. The current viewing position is not affected by this command. See also “printline” on page 119.

This command is typically used in non-GUI mode.

pwd

pwd

Prints MULTI's current working directory.

By default, this is the directory in which your program runs. However, this may be overridden by the **rundir** command or by its GUI equivalent, the **Start in** field of the **Arguments** dialog box. See “rundir” on page 174 and “The Arguments Dialog Box” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book.

Q

Q [0 | 1 | b]

Sets or toggles the Debugger's quiet mode. The command `Q 0` turns off quiet mode (the default for quiet mode is off), the command `Q 1` turns on quiet mode, and the command `Q` alone toggles quiet mode. When the Debugger is in quiet mode, many commands are less verbose. For example, when a breakpoint is being set or toggled in quiet mode, the breakpoint will not be echoed to the command pane.

The command `Q b` is valid only in a breakpoint's command list. When the process stops at a breakpoint that contains `Q b` in its command list, the Debugger will not print a message about the breakpoint being hit (Stopped by breakpoint).

savedebugpane

savedebugpane [[cmd]][[target]][[io]][[serial]][[python]][[traffic]] ["filename"]

GUI only

Saves the contents of the specified Debugger pane to the file *filename*. If you do not specify *filename*, a **Save As** dialog box prompting you to choose a file appears. There are six possible panes: the command pane, the target pane, the I/O pane, the serial terminal pane, the Python pane, and the traffic pane. If you do not specify a pane, the contents of the currently visible pane are saved. See also “debugpane” on page 111.

windowcopy

windowcopy wid=*num*

GUI only

Copies the current selection in the window specified by the window identification number, *num*, to the clipboard. For more information about window identification numbers, see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

windowpaste, windowspaste

windowpaste wid=*num*

windowspaste wid=*num*

GUI only

Pastes the current selection into the input buffer of the window whose identification number is *num*. This command is typically used as part of a **mouse** or **keybind** command. For more information about the **mouse** and **keybind** commands and window identification numbers, see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

The **windowspaste** command uses the selection, whereas the **windowpaste** command uses the clipboard.

Chapter 9

Help and Information Command Reference

Contents

Help and Information Commands	124
-------------------------------------	-----

The commands in this chapter allow you to access help information or information about your MULTI installation.

Help and Information Commands

The following list provides a brief description of each help and information command. For a command's arguments and for more information, see the page referenced.

- **about** — Displays information about MULTI (see “about” on page 124).
- **aboutlic** — Displays information about the licenses in use by MULTI (see “aboutlic” on page 125).
- **bugreport** — Launches the **gbugrpt** utility, which allows you to append displayed information to a bug report form that you can fill out and email to Green Hills Software's Technical Support (see “bugreport” on page 125).
- **help** — Displays or searches for documentation in the Help Viewer (see “help” on page 125).
- **info** — Prints information about the state of MULTI (see “info” on page 125).
- **usage** — Prints the syntax for the specified Debugger command (see “usage” on page 126).

about

about

Displays information about MULTI.

In GUI mode, this command opens a dialog box that contains information about the current version of MULTI. In non-GUI mode, this command prints this information to standard output.

Corresponds to: **Help** → **About MULTI**

aboutlic

aboutlic

GUI only

Displays information about the licenses in use by MULTI.

Corresponds to: **Help** → **License Info**

bugreport

bugreport

GUI only

Launches the **gbugrpt** utility, which displays information about your MULTI installation and allows you to append it to a bug report form that you can fill out and email to Green Hills Software's Technical Support.

help

help [*keyword* | *command* | *configuration_option*]

Opens the Help Viewer on documentation for the MULTI *command* or *configuration_option* or, if a keyword is specified, searches all manuals for *keyword*. If no argument is specified, the *MULTI: Debugging* book opens in the Help Viewer.

info

info

Prints the following information about the state of MULTI:

- Debugging status
- Current target connection
- On Linux/Solaris, core file status

- Child program status
- Output recording status
- Command recording status
- Case sensitivity status

usage

usage *command*

Prints the syntax for the specified MULTI Debugger command.

The conventions for displaying the command syntax in the command pane are similar to those used in this book (see “Conventions Used in the MULTI Document Set” on page xviii), with the following exception: In the syntax returned by the **usage *command*** command, words that appear in all capital letters are placeholders and should be replaced in your actual command line with a value appropriate for your context. (These placeholders appear in italics, rather than capital letters, in the print and online documentation.) For example, *FILENAME* indicates that you should substitute the name of the file to be used for the operation performed by the command.

Chapter 10

Memory Command Reference

Contents

General Memory Commands	128
-------------------------------	-----

The commands in this chapter allow you to perform memory-related operations such as filling, copying, or writing to a specified block of memory and performing memory reads and tests. See also “Cache View Commands” on page 305.

General Memory Commands

The following list provides a brief description of each general memory command. For a command's arguments and for more information, see the page referenced. (Note that complete descriptions for some of these commands are located in other chapters.)

- **compare, compareb** — Compares two blocks of memory (see “compare, compareb” on page 129).
- **copy, copyb** — Copies one region of memory to another (see “copy, copyb” on page 130).
- **disassemble** — Disassembles a specified region of memory (see “disassemble” on page 131).
- **fill, fillb** — Fills a specified block of target memory (see “fill, fillb” on page 132).
- **find, findb** — Searches for a block of memory (see “find, findb” on page 133).
- **flash** — Writes a file to flash memory on the target (see “flash” on page 134).
- **memdump** — Copies a section of memory on the target to a specified file on the host (see “memdump” on page 136).
- **memload** — Loads the contents of a file on the host machine into a portion of target memory (see “memload” on page 137).
- **memread** — Performs a sized memory read from the target and prints the result (see “memread” on page 139).
- **memtest** — Configures and launches memory tests (see “memtest” on page 139).
- **memview** — Opens a **Memory View** window for displaying and modifying memory contents (see “memview” on page 299 in Chapter 21, “View Command Reference” on page 291).
- **memwrite** — Performs a sized memory write to the target (see “memwrite” on page 142).

- **verify** — Verifies that the contents of memory match the contents of the executable program file (see “verify” on page 143).

compare, compareb

compare -gui

compareb -gui

compare [-noprogress] [*operation*] *src1 src2 length* [*size*]

compareb [-noprogress] [*operation*] *src1 src2 bytes* [*size*]

The -gui argument to the **compare** or **compareb** command opens a window where you can enter the parameters for comparing two blocks of memory.

The **compare** -gui command corresponds to **Target → Memory Manipulation → Compare** and to the **Memory View** menu selection **Memory → Compare**.

The second format of the **compare** and **compareb** commands compares the elements of two regions of memory beginning at the addresses *src1* and *src2*. The size of the block of memory that is compared is determined by *length* for the **compare** command and *bytes* for the **compareb** command.

Corresponding elements from the two locations are each compared in turn, using the given *operation* (see below), where each element is *size* bytes long. For the **compare** operation, the total size in bytes of the blocks compared is (*length* × *size*), and for the **compareb** operation it is *bytes*. The argument *size* may be either 1, 2, 4, or 8 bytes. If *size* is not specified, the default is the size of an integer on the target system.

The argument *operation* may be any of the following values:

- <= — The element in *src1* is less than or equal to the element in *src2*.
- < — The element in *src1* is less than the element in *src2*.
- >= — The element in *src1* is greater than or equal to the element in *src2*.
- > — The element in *src1* is greater than the element in *src2*.
- == — The element in *src1* is equal to the element in *src2*.
- != — The element in *src1* is not equal to the element in *src2*.

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

A progress bar is shown for the comparison unless `-noprogess` is specified.

The **compare** and **compareb** commands print each pair of corresponding elements that have the relationship described by *operation*. The following example compares two overlapping arrays of six 4-byte integers. The first array starts at 0x10000, and the second at 0x10008. The **compare** command displays only the pairs that satisfy *operation*.

```
> compare >= 0x10000 0x10008 6 4
0x10000, 0x10008 : 2091264888, 2086935416
0x10004, 0x1000c : 2089100152, 945815572
0x10008, 0x10010 : 2086935416, 1279398274
0x10014, 0x1001c : 1207968893, 1099038740
```

These commands require that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

To interrupt these commands, press **Esc**.

copy, copyb

copy -gui

copyb -gui

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

copyb *src dest bytes [size]*

The `-gui` argument to the **copy** or **copyb** command opens a window where you can enter the parameters for copying one region of memory to another.

The **copy** -gui command corresponds to **Target** → **Memory Manipulation** → **Copy** and to the **Memory View** menu selection **Memory** → **Copy**.

The second format of the **copy** and **copyb** commands copies a block of memory with elements of size *size* from *src* to *dest*. For the **copy** command, the block consists of *length* elements of size *size*. Thus, the total size of memory copied in bytes is (*length* × *size*). For the **copyb** command, the total size of the block

is *bytes*. The argument *size* may be either 1, 2, 4, or 8 bytes. If *size* is not specified, it defaults to the size of an integer on the target system. For the **copy** command, the direction of the copy may be specified by *direction*, and may be either *forw* for forward copying [default], or *backw* for reverse copying.

Reverse copying is the same as forward copying, except that the elements at the end of the block are written first, before the elements at the beginning of the block. Reverse copying will have the same effect as forward copying, unless the *src* and *dest* regions overlap.

These commands require that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

To interrupt these commands, press **Esc**.

disassemble

disassemble [-quiet] *addr_expr* [*size*]

disassemble [-quiet] -section *section*

disassemble [-quiet] -recheck

Disassembles a specified region of memory and adds it to MULTI's cache. By default, the disassembled instructions are displayed in the command pane. Available options are:

- *-quiet* — Specifies that disassembled instructions should not be displayed in the command pane.
- *addr_expr* [*size*] — Specifies the region of memory to disassemble, beginning at *addr_expr* and ending *size* bytes later. However, if the last instruction of the region continues past *size* bytes, additional memory may be read to disassemble the full instruction. *addr_expr* may be any expression that represents a memory address. If *size* is not specified, MULTI disassembles the region of memory starting at *addr_expr* and ending at the end of the containing function.
- *-section section* — Specifies the section of memory to disassemble.
- *-recheck* — Performs the last disassembly again.

fill, fillb

fill -gui

fillb -gui

fill [-noprogress] *dest length* [*value* [*size*]]

fillb [-noprogress] *dest bytes* [*value* [*size*]]

The **-gui** argument to the **fill** or **fillb** command opens a window where you can enter the parameters for filling a specified block of memory.

The **fill** -gui command corresponds to **Target → Memory Manipulation → Fill** and to the **Memory View** menu selection **Memory → Fill**.

The second format of the **fill** and **fillb** commands fills the target's memory with the given value. For the **fill** command, the block starts at *dest* and extends for *length* elements of *size* bytes. Thus, the total size of the block in bytes is (*length* × *size*). For the **fillb** command, the block has a length in bytes of *bytes*. The block will be filled with *value*, or zero if *value* is not specified. The argument *size* is the number of bytes to place *value* in and may be either 1, 2, 4, or 8. If *size* is not specified, the default is the size of an integer on the target system. If *value* is too large to fit in the elements of the size given, the most significant bits of *value* are truncated.

A progress bar is shown for the second format of the **fill** and **fillb** commands unless **-noprogress** is specified.

These commands require that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

To interrupt these commands, press **Esc**.

find, findb

find -gui

findb -gui

find [-noprogess] *src length value [size [mask]]*

findb [-noprogess] *src bytes value [size [mask]]*

The **-gui** argument to the **find** or **findb** command opens a window where you can enter the parameters for searching for a value in memory.

The **find** -gui command corresponds to **Target → Memory Manipulation → Find**.

The **findb** -gui command corresponds to the **Memory View** menu selection **Memory → Find**.

The second format of the **find** and **findb** commands searches memory starting at *src* for an element that is *size* bytes long and has the given *value*. The argument *size* may be 1, 2, 4, or 8 bytes and, if not specified, is the size of an integer on the target system. For the **find** command, the search stops when *length* elements have been checked. For the **findb** command, the search stops when *bytes* bytes have been checked. If *mask* is specified, it is logically ANDed with each memory location value before being compared with *value*. Every match found is listed on a separate line with the address of the match.

A progress bar is shown for the second format of the **find** and **findb** commands unless **-noprogess** is specified.

These commands require that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

To interrupt these commands, press **Esc**.

flash

flash *gui* [*program_name*]

flash *burn* [-baseaddress=*address*] [-type= *elf* | *raw* | *srec*] [-offset=*offset*]
[-executable=*filename*] [-endian= *big* | *little* | *auto*] [-rambase=*address*]
[-script=*filename*] [-eraseonly= *yes* | *no*] [-unlock= *yes* | *no*] [-verify= *yes* | *no*]
[-memrequire=*kilobytes*]

The *gui* argument to the **flash** command opens a window where you can enter the parameters for writing a file to flash memory on the target. The available option is:

- *program_name* — Opens this window on the program specified. If you do not specify a *program_name*, this window opens on the program you are debugging.

The **flash** *gui* command corresponds to **Target** → **Flash**.

See also “prepare_target” on page 253.

The *burn* argument to the **flash** command causes the **MULTI Fast Flash Programmer** to write a file to flash memory on the target, where:

- -baseaddress=*address* — Specifies the base address of the flash memory in the target memory map. *address* can be in decimal or hex format. This defaults to the last value you specified for this executable. If no previous value is available, 0 is used.
- -type=*elf* | *raw* | *srec* — Identifies the file format of the file to be written to flash memory (that is, the file specified via the -executable argument). Use *elf* if the file format is ELF. Use *raw* if the file is an unformatted memory image. Use *srec* if the file is a collection of S-Records. This defaults to *elf*.
- -offset=*offset* — Specifies an offset for the write. If the file is a memory image, the offset is the location relative to the base address where it will be written. If the file is an ELF, S-Record, or MULTI internal format program, the offset is added to the addresses of each section. This defaults to 0.
- -executable=*filename* — Specifies the path on the host of the file to be written to flash memory. This defaults to the program you are currently debugging.

- `-endian=big|little|auto` — Specifies the endianness of the target CPU. This defaults to `auto`, which causes the **MULTI Fast Flash Programmer** to read the setting from the debug adapter.
- `-rambase=address` — Sets the location in target RAM where the **MULTI Fast Flash Programmer** will temporarily store agents and other data. This defaults to the RAM base address defined in the executable's linker directives file. If the RAM base address cannot be determined, the last value you specified for this executable is used; otherwise, 0 is used.
- `-script=filename` — Specifies the path to a setup script that will be run before the flash programming session. This defaults to the setup script path (if any) provided in the target's connection settings.
- `-eraseonly=yes|no` — This disables the programming function of the **MULTI Fast Flash Programmer**, which causes the sectors covered by the input file to be left in the erased state. This defaults to `no`.
- `-unlock=yes|no` — Specifies whether locked sectors should be unlocked before programming. This defaults to `no`. Reprogramming locked sectors may change the boot sequence of the target board.
- `-verify=yes|no` — Enables or disables the verification stage of flash programming. This defaults to `yes`.
- `-memrequire=kilobytes` — Specifies, in kilobytes, the amount of target RAM to be used for target agents and other data. Generally, increasing this value results in faster flash programming. This defaults to all available RAM (up to 2 MB) as defined in the executable's linker directives file. If the amount of available RAM cannot be determined, 128 KB is used.

This command requires that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

For more information, see Chapter 26, “Programming Flash Memory” in the *MULTI: Debugging* book.

memdump

memdump -gui

memdump [-append] [-noprogess] [srec | raw] *filename start length*

memdump [-append] [-noprogess] [srec | raw] *filename section*

The `-gui` argument to the **memdump** command opens a window where you can enter the parameters for copying a block of memory on the target to a file on the host.

The **memdump** -gui command corresponds to **Target → Memory Manipulation → Memory Dump** and to the **Memory View** menu selection **Memory → Dump**.

The second and third formats of the **memdump** command copy a section of memory on the target to a file on the host, where:

- `-append` — Adds data to the end of the file rather than overwriting existing data in the file.
- `-noprogess` — Does not show a progress bar for the operation.
- `srec` — Specifies Motorola S-Record format.
- `raw` — Specifies raw binary data.
- *filename* — Specifies the file that memory is copied to.
- *start* — Specifies the starting address in memory to dump.
- *length* — Specifies how many bytes of data to dump to the file, beginning at *start*.
- *section* — Specifies the name of a section in memory to dump.

Both the **memload** and **memdump** commands support these file formats: S-Record, raw, and default. If you do not specify `srec` or `raw`, the default format is used. For a detailed description of these formats, see “memload” on page 137.

This command requires that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

To interrupt this command, press **Esc**.

memload

memload -gui

memload [srec | raw | legacy] [-w *size*] [-noprogress] *filename* [-offset *file_offset*]
[*start* [*length*] | *section*]

The `-gui` argument to the **memload** command opens a window where you can enter the parameters for loading the contents of a file on the host machine into a portion of memory on the target.

The **memload -gui** command corresponds to **Target → Memory Manipulation → Memory Load** and to the **Memory View** menu selection **Memory → Load**.

The second format of the **memload** command loads the contents of a file on the host machine into a portion of target memory, where:

- *srec* — Specifies Motorola S-Record format.
- *raw* — Specifies raw binary data.
- *legacy* — Specifies that MULTI should use the default format written by MULTI 4 and earlier. This option is only available from the command line.
- `-w size` — Specifies the size (in bytes) of the individual memory writes. The value must be 1, 2, or 4. The default, which depends on your target hardware and debug server, is selected to optimize loading performance.
- `-noprogress` — Does not show a progress bar for the operation.
- *filename* — Specifies the file to load into memory.
- `-offset file_offset` — Specifies a number of bytes at the beginning of the file to skip before performing the memory load. This option is only applicable if specified in conjunction with *raw*.
- *start* — Specifies the starting address in memory to load.
- *length* — Specifies how many bytes of data to load into memory, beginning at *start*. The *length* argument must be a multiple of *size*.
- *section* — Specifies the name of a section in memory to load.

Both the **memload** and **memdump** (see “memdump” on page 136) commands support these file formats: S-Record, raw, and default.

To use the S-Record format, specify the `srec` option. If the arguments `size`, `start`, `length`, or `section` are specified, they are ignored. The file is read as a Motorola S-Records file.

To use the raw format, specify the `raw` option. In this format, the file contains only the binary data, with no formatting or header information. This format is appropriate for transferring data to external tools that deal with binary data. You must specify `start` and optionally `length`, or you must specify `section`. This command loads `filename`, starting at the first byte of the file (unless `-offset file_offset` is specified) and continuing for `length` bytes. To skip a number of bytes at the beginning of the file, use the `-offset file_offset` option. If `length` is not specified, this command will load the entire content of the file (without `-offset file_offset`) or the remainder of the file (with `-offset file_offset`).

To use the legacy format, specify the `legacy` option. This loads default format files that were created by MULTI 4 or earlier.

To use the default format, do not specify the `srec`, `raw`, or `legacy` option. This file format is proprietary and non-portable. The address (8 bytes) and size (4 bytes) of the memory region is stored at the beginning of the file in host byte order. The actual content of the memory region follows the address and size, and is handled as a series of bytes. If `start`, `length`, and `section` are omitted, the values specified when the file was created are used.

If MULTI is always run on a single host, the default format is the easiest to use when dumping memory to be read back into MULTI. Because the **memdump** command records the address and size in the file (see “memdump” on page 136), it is not necessary to specify `start`, `length`, or `section` when using the **memload** command to load a file into memory that was created with **memdump**.

This command requires that MULTI be connected to a target and that the target be in a state such that MULTI can access memory.

To interrupt this command, press **Esc**.

memread

memread *size address*

Performs a sized memory read from the target and prints the result. This command is intended to be used to perform low-level reads from 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 also the system variable `_CACHE` in “System Variables” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

size may be 1, 2, or 4. Some targets, such as the Green Hills Probe, INTEGRITY IoT, and INTEGRITY version 10 and later, also support 8. The units are bytes. *address* must be aligned correctly to the nearest *size* bytes.

memtest

memtest *start_addr end_addr* -size=*size* -test=*test_choice* [-pattern=*value*] [-complement | -rotate | -random | -complement -rotate] [-maxtransitions] [-resetpattern] [-repeat=*number_of_tests* | -continuous] [-maxerr=*number_of_errors* | -writeonly] [-tgtagent] [-tgtagentstart | -tgtagentend | -tgtagentloc=*expr*]

Configures and launches memory tests, where:

- *start_addr* — Defines the lowest address to test. *start_addr* must be an expression that evaluates to a 32-bit address.
- *end_addr* — Defines the highest address to test. *end_addr* must be an expression that evaluates to a 32-bit address.
- *size* — Indicates the access size (in bytes) to use when performing the test. Valid sizes are 1, 2, and 4.
- -test=*test_choice* — Defines the type of test to run. Acceptable values for *test_choice* are:
 - a1 — Address walking one test (destructive)
 - a0 — Address walking zero test (destructive)
 - d1 — Data walking one test (destructive)
 - d0 — Data walking zero test (destructive)
 - p — Data pattern test (destructive)

- `r` — Memory read test (nondestructive)
- `cr` — CRC computation (nondestructive)
- `cc` — CRC compare test (nondestructive)
- `fr` — Find start/end ranges test (nondestructive)

More than one of the destructive memory test options (`a1`, `a0`, `d1`, `d0`, and `p`) can be specified by using multiple `-test=test_choice` arguments. The nondestructive tests (`r`, `cr`, `cc`, and `fr`) must be performed individually. For descriptions of these tests, see “Types of Memory Tests” in Chapter 29, “Testing Target Memory” in the *MULTI: Debugging* book.

- `-pattern=value` — Specifies the data value to use for address bus walking and/or data pattern tests.
- `-complement` — Causes the data pattern value for pattern tests to be complemented between memory writes. (This option can be passed with the `-rotate` option. If both `-complement` and `-rotate` are specified, the pattern will be rotated every other write and complemented every write, resulting in a write sequence similar to `0x01`, `0xfe`, `0x02`, `0xfd`, ...))
- `-rotate` — Causes the data pattern for pattern tests to be rotated between writes. (This option can be passed with the `-complement` option as well, causing the pattern value to be both complemented and rotated between iterations. See the description for `-complement` above.)
- `-random` — Causes a pseudorandom sequence of values to be used for the pattern test. (This option cannot be used with either the `-complement` or the `-rotate` options.)
- `-maxtransitions` — Causes MULTI to use a sequence of addresses in the data pattern or memory read tests that maximizes the address line transitions between accesses. (The default behavior is to access memory sequentially from low addresses to high addresses.)
- `-resetpattern` — Causes MULTI to use the same starting pattern value for each test iteration rather than using a complemented, rotated, or subsequent pseudorandom value.
- `-repeat=number_of_tests` — Specifies the number of times to repeat a test or tests. (This option cannot be passed with the `-continuous` option, and does not apply to the CRC compute or find start/end ranges tests.)

If a `-repeat` value is specified with multiple memory tests, all selected tests are run during each iteration. For example, the options `-test=a0 -test=p -repeat=2` would set the test sequence as:

```
Address walking zero
Pattern
Address walking zero
Pattern
```

- `-continuous` — Specifies that the test(s) will run continuously. (This option cannot be passed with the `-repeat=number_of_tests` option, and does not apply to the CRC compute or find start/end ranges tests.)
- `-maxerr=number_of_errors` — Causes MULTI to abort the test(s) after detecting the specified number of errors. This option cannot be used with the `-writeonly` option.
- `-writeonly` — Causes MULTI to skip the reading phase of the address bus walking, data bus walking, and data pattern tests. This option cannot be used with the `-maxerr=number_of_errors` option.
- `-tgtagent` — Causes MULTI to download and use a target-resident agent to perform the memory tests (rather than using the Debugger).
- `-tgtagentstart` — Specifies that the target agent be placed at the start of the memory range to be tested. This option is only valid when used with the `-tgtagent` option.
- `-tgtagentend` — Specifies that the target agent be placed at the end of the memory range to be tested. This option is only valid when used with the `-tgtagent` option.
- `-tgtagentloc=expr` — Specifies that the target agent be placed at the location specified by the expression `expr`. This memory location should not overlap the test range, and must be valid for the test to be performed successfully.

For further details about test types and options, see “Advanced Memory Testing: Using the Perform Memory Test Window” in Chapter 29, “Testing Target Memory” in the *MULTI: Debugging* book and “Types of Memory Tests” in Chapter 29, “Testing Target Memory” in the *MULTI: Debugging* book.

**Note**

The **memtest** command syntax can be complex, because of the variety of test types and options available. For this reason, we recommend that you use the **Memory Test Wizard** or the **Perform Memory Test** window to configure and run memory tests. Even if you want to write scripts that contain memory testing commands, you can still use the interactive interfaces to determine the exact command syntax for the specific testing options you want to use. To do this, use one of these GUIs to configure the test you want to run, and then run the test. When the test completes, the **Memory Test Results** window will display the exact command syntax that corresponds to the testing options you specified. You can use the command syntax given there from the Debugger command pane or in scripts you write. For more information, see “Advanced Memory Testing: Using the Perform Memory Test Window” in Chapter 29, “Testing Target Memory” in the *MULTI: Debugging* book and “Viewing Memory Test Results” in Chapter 29, “Testing Target Memory” in the *MULTI: Debugging* book.

memwrite

memwrite *size address value*

memwrite -str *address "string"*

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. This command has two possible formats.

In the first format, without -str specified, **memwrite** performs a sized memory write, where:

- *size* — Specifies the access size in bytes and may be 1, 2, or 4. Some targets, such as the Green Hills Probe, INTEGRITY IoT, and INTEGRITY version 10 and later, also support 8.
- *address* — Specifies the memory address at which to begin writing. *address* must be aligned correctly to the nearest *size* bytes.

- *value* — Specifies the value to be written. If *value* is too large to fit in *size* bytes, it is truncated to fit.

In the second format, with `-str` specified, **memwrite** writes the specified string to the target, including the terminating null character, where:

- *address* — Specifies the memory address at which to begin writing.
- "*string*" — Specifies a string constant to be written to the target. The quotation marks are part of the syntax and must appear around *string*.

verify

verify [-quiet] [-sparse] -all | -recheck | -section *section_name* | *address_expression* [*num_addresses*]

Verifies that the contents of memory match the contents of the executable program file. If there are any coherency errors (that is, discrepancies), the associated lines are highlighted in the source pane, and a list of differing addresses are printed to the command pane (both the in-memory and the executable program file values are shown). A progress bar displays the work done; **Esc** aborts processing.

Available options are:

- `-quiet` — Highlights lines associated with coherency errors but does not print output to the command pane. Highlighting is done in the source pane.
- `-sparse` — Verifies only a few bytes at the beginning, middle, and end of the indicated range. This allows you to run a quick coherency scan rather than a complete test. This option has no meaning and is ignored if specified in conjunction with `-recheck`.
- `-all` — Verifies all downloaded non-data sections that cannot be written to. The `.text` section is an example of one such section. This may take a long time.

Because certain sections of memory, such as `.data`, may be written to during program execution, you can expect them to differ from the executable program file. When you specify this option, the **verify** command does not check these sections. However, you can verify them manually by specifying `verify -section section_name`.

- `-recheck` — Re-examines any addresses that were previously found to contain coherency errors.
- `-section section_name` — Verifies the section `section_name`. Only sections whose contents exist in the executable file can successfully be verified.
- `address_expression [num_addresses]` — Verifies the block of memory starting at the address expression `address_expression` and continuing for `num_addresses`, where `num_addresses` is the number of addresses to verify. The maximum supported value of `num_addresses` is `0x7fffffff`. If you omit `num_addresses`, this command verifies until the end of the function that encloses `address_expression`. Note that only those addresses corresponding to memory whose contents exist in the executable file can successfully be verified; other regions will be ignored.

You can verify sections from any module loaded into target memory (including shared objects such as `libc.so`). However, if multiple identically named sections exist, `verify -section section_name` verifies only the first `section_name` it finds. For example, if `.text` appears first in a shared object and then in an executable program file, `verify -section .text` finds `.text` only in the shared object. To more accurately specify sections, use the **map** command to determine the starting address and size of the section you want to verify (see “map” on page 117).

For more information about coherency errors, see “Detecting Coherency Errors” in Chapter 29, “Testing Target Memory” in the *MULTI: Debugging* book.

See also “prepare_target” on page 253.

Chapter 11

Navigation Command Reference

Contents

Navigation Commands	146
---------------------------	-----

The commands in this chapter allow you to navigate the code displayed in the Debugger source pane. See also Chapter 16, “Search Command Reference” on page 235.

Navigation Commands

The following list provides a brief description of each navigation command. For a command's arguments and for more information, see the page referenced.

- **+** — Moves your current viewing position in the source pane the specified number of lines toward the end of the file (see “+” on page 147).
- **-** — Moves your current viewing position in the source pane the specified number of lines toward the beginning of the file (see “-” on page 147).
- **e** — Navigates around your program in the Debugger's source pane (see “e” on page 147).
- **indexnext** — Changes the current viewing location to the next item in Debugger's history list (see “indexnext” on page 148).
- **indexprev** — Changes the current viewing location to the previous item in the Debugger's history list (see “indexprev” on page 149).
- **number** — Moves your current viewing position in the source pane to the file-relative line number specified (see “number” on page 149).
- **scrollcommand** — Scrolls the given window by the specified amount and in the specified direction (see “scrollcommand” on page 149).
- **switch** — Changes the selection in the target list, and optionally opens a new Debugger window on the selection or brings an existing new window to the front (see “switch” on page 151).
- **uptosource** — Displays the first function on the stack that contains source code (see “uptosource” on page 151).

+

+ [*num*]

GUI only

Moves your current viewing position in the source pane *num* lines toward the end of the file. The default value for *num* is 1 line.

-

- [*num*]

GUI only

Moves your current viewing position in the source pane *num* lines toward the beginning of the file. The default value for *num* is 1 line.

e**e** [*address_expression*]

Navigates around your program in the Debugger's source pane. You can also use this command to open a Browse window. This is one of the most powerful and commonly used navigation features of the MULTI Debugger.

If *address_expression* is specified, the **e** command changes your current viewing location in the code to that address expression. See “Using Address Expressions in Debugger Commands” on page 5. With no arguments, this command prints your viewing location in the code. There are several forms of the **e** command.

- **e** — Shows current file, function, line number, and address. For example:
test.c:PrintLine#28: 0x411c
- **e func** — Displays *func* if there is an exact match for it. *func* may contain wildcards. Note that if *func* matches the name of a file and a function, the file is displayed, not the function. If there are no matches for *func*, the **e** command operates on *func**. If there are multiple matches, a **Browse** window opens to display all of them.

- **e "file"#func** — Displays *func* located in *file* if there is an exact match for it. *func* and *file* may contain wildcards. If there are no matches for *func/file*, the **e** command operates on *func*/file**. If there are multiple matches, a **Browse** window opens to display all of them.
- **e file** — Displays *file* if there is an exact match for it. *file* may contain wildcards. If there are no matches for *file*, the **e** command operates on *file**. If there are multiple matches, a **Browse** window opens to display all of them. Note that if *file** matches the names of multiple files and multiple functions, the **Browse** window displays only the function names, not the file names.
- **e variable** — Displays the source location where the variable *variable* is defined. This is only valid for variables with unique names that have absolute locations (i.e., global and file static variables).
- **e num_** — Displays the function at call stack frame number *num*. The call stack frame number must be followed by an underscore (_). Use the **calls** command to view the entire call stack (see “calls” on page 78).
- **e address_expression** — Displays the function at the given address. See “Using Address Expressions in Debugger Commands” on page 5.
- **e +offset** — Displays the source associated with the instructions that are *offset* source lines after the currently viewed location.
- **e -offset** — Displays the source associated with the instructions that are *offset* source lines before the currently viewed location.
- **e numb** — Displays the function containing breakpoint number *num*. For example, **e 1b** displays the function containing breakpoint number one. You can use the **B** command to view breakpoint numbers (see “B” on page 48).

Corresponds to: **View** → **Navigation** → **Goto Location**

indexnext

indexnext

GUI only

Changes the current viewing location to the next item in Debugger's history list. See also “Using Navigation History Buttons” in Chapter 9, “Navigating Windows and Viewing Information” in the *MULTI: Debugging* book.

Corresponds to: 

indexprev

indexprev

GUI only

Changes the current viewing location to the previous item in the Debugger's history list. See also “Using Navigation History Buttons” in Chapter 9, “Navigating Windows and Viewing Information” in the *MULTI: Debugging* book.

Corresponds to: 

number

number

GUI only

Moves your current viewing position in the source pane to the file-relative line number *number*. This command ignores the option **Use function relative line numbers (vs. file relative) (procRelativeLines)**.

scrollcommand

scrollcommand [+ | -] max[1 | c] [wid=*num*]

scrollcommand [+ | -] page[1 | c] [wid=*num*]

scrollcommand [+ | -] *count* [1 | c] [wid=*num*]

GUI only

Scrolls the window indicated by the identification number *num* by the specified amount and in the specified direction.

With Format 1, the window is scrolled completely to the beginning ($-max$) or the end of its view ($+max$ or max).

With Format 2, the window is scrolled to the previous page (`-page`) or the next page (`+page` or `page`).

With Format 3, the window is scrolled by the `count` number of lines or characters, where `count` may be positive or negative.

If `count`, `max`, or `page` is followed by the letter `c`, the scroll is horizontal and `count` corresponds to the number of characters.

If `count`, `max`, or `page` is followed by the letter `l` (lowercase `L`), the scroll is vertical and `count` corresponds to the number of lines. Vertical scrolling is the default, if neither `c` nor `l` is specified.

The window identification number `num` is obtained by using the special sequence `%w` with either the **mouse** command or the **keybind** command (see the **keybind** and **mouse** commands in “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book). If no window identification number is specified, the source window is used.

For example, the following command scrolls the source pane one line towards the end of the file:

```
> scrollcommand 1
```

The following scrolls the pane that is currently visible in the Debugger (for example, the command pane) backwards by two lines:

```
> scrollcommand -2 wid=-2
```

The following scrolls the source pane three characters to the right:

```
> scrollcommand 3c
```

Both of the following commands scroll the source pane to the beginning of the code:

```
> scrollcommand -max  
> scrollcommand -maxl wid=-1
```

switch

switch -direction up | -direction down | -direction up+ | -direction down+ | -selectall | -component *component_name* | -item *item_prefix* [-new_window]

Changes the selection in the target list, and optionally opens a new Debugger window on the selection or brings an existing new window to the front. The arguments for this command are:

- -direction up|down — Moves the target list selection up or down one entry.
- -direction up+|down+ — Extends the target list selection up or down one entry.
- -selectall — Selects all entries in the target list.
- -component *component_name* — Selects the entry in the target list that matches the specified component, if possible. Some components may not be supported. For information about listing components, see “components” on page 110.
- -item *item_prefix* — Selects the first entry that matches the given prefix string. Note that prefixes with spaces must be quoted.
- -new_window — Opens a new Debugger window on the entry specified by -component *component_name* or -item *item_prefix*, or brings the existing new window to the front. -new_window cannot be used in conjunction with the other arguments to this command.

uptosource

uptosource

Displays the first function on the stack that contains source code. This command does not change the program counter or execute any program instructions on the target.

Corresponds to: **View → Navigation → UpStack To Source**

Chapter 12

Profiling Command Reference

Contents

profile	154
---------------	-----

The command in this chapter allows you to control MULTI's profiling capabilities, enable the collection of profiling data, open the **Profile** window, and access profiling information.

Unless otherwise stated, the button and menu choices listed alongside the **profile** command options are displayed in the **Profile** window.

For more information about profiling, see Chapter 17, “Recording and Viewing Profiling Data” in the *MULTI: Debugging* book.

For information about the **protrans** shell command, which is used to invoke the **protrans** utility, see the documentation about the protrans utility in the *MULTI: Building Applications* book.

profile

```
profile [ start | stop | process ] [mode ( performance | coverage )] [import64 filename]
[select ( all | none ) functions ] [filter ( save | load ) filename ] [save filename] [load
filename] [print ( tasks | files | functions | functions_source | coverage |
coverage_source | blocks )] [collect filename] [traceimport] [csvexport [tasks
filename] [files filename] [functions filename] [functions_source filename] [coverage
filename] [coverage_source filename] [blocks filename]] [display ( none | percent
| coverage )] [clear] [close]
```

The above parentheses are used for grouping; they are not a part of the command.

GUI only

Enables the collection of profiling data and configures MULTI's profiling capabilities. Available arguments are:

- *no argument* — Opens the **Profile** window. This option corresponds to **View** → **Profile** in the Debugger.
- *start* — Starts a new recording in the **Profile** window. This option corresponds to the **Start** button.
- *stop* — Stops a recording in the **Profile** window and processes the profiling data. This option corresponds to the **Stop** button.

- `process` — Collects instrumented profiling data from the target, adds an entry for it in the **Profile** window's **Available Recordings** list, and loads it into the display. This option corresponds to the **Process** button.
- `mode` — Opens the **Profile** window in one of the following view modes, or resets the view mode in the existing **Profile** window to one of the following values:
 - `performance` — Displays PC sampling information, History information, or trace information in performance mode (corresponds to **Mode: PC Sampling**, **Mode: Performance (History)**, and **Mode: Performance (Trace)**, respectively).
 - `coverage` — Displays coverage information generated from trace data (corresponds to **Mode: Coverage**), or displays instrumentation data, which includes instruction-level performance information and/or coverage (**Mode: Instrumentation**).
- `import64 filename` — Imports a **bmon64.out** file into a new recording. The Debugger must be open on the executable for which the profile data is being imported, and that executable must be selected in the target list.
- `select` — Adds or removes functions from the filter:
 - `all functions` — Adds all functions.
 - `none functions` — Removes all functions.
- `filter` — Saves or loads the filter:
 - `save filename` — Saves the filter to the specified file. The default file extension is **.gprf**.

If you subsequently modify your program and then load the saved filter, any new items in your program are checked in the filter unless a parent item was cleared. For example, a new file will be checked unless the containing directory was cleared in the saved filter.

- `load filename` — Loads the filter from the specified file.
- `save filename` — Saves the current recording into a **.gpro** file.
- `load filename` — Loads the **.pro**, **.gpro**, or **bmon.out** file *filename* into a new recording. If you are loading a **bmon.out** or **.pro** file, the Debugger must be open on the executable for which the profile data is being imported, and that executable must be selected in the target list.

- `print` — Prints the current profiling view as an ASCII table:
 - `tasks` — Prints the contents of the **Tasks** tab or **AS** tab.
 - `files` — Prints the contents of the **Files** tab.
 - `functions` — Prints the contents of the **Functions** list on the **Functions** tab.
 - `functions_source` — Prints the contents of the **Source Lines** list on the **Functions** tab.
 - `coverage` — Prints the contents of the **Functions** list on the **Coverage** tab.
 - `coverage_source` — Prints the contents of the **Source Lines** list on the **Coverage** tab.
 - `blocks` — Prints the contents of the **Blocks** list on the **Blocks** tab.
- `collect filename` — Merges instrumented profiling data stored on the target into the specified **.gpro** recording.
- `traceimport` — Imports the contents of the current trace data into a new recording. The Debugger is suspended until trace processing is complete.
- `csvexport` — Exports the current profiling view as one or more **.csv** files. You can specify one or more of the following arguments:
 - `tasks filename` — Exports the contents of the **Tasks** or **AS** tab to *filename*.
 - `files filename` — Exports the contents of the **Files** tab to *filename*.
 - `functions filename` — Exports the contents of the **Functions** list on the **Functions** tab to *filename*.
 - `functions_source filename` — Exports the contents of the **Source Lines** list on the **Functions** tab to *filename*. This exports source line information for all visible functions, not just the source lines visible in the **Profile** window.
 - `coverage` — Exports the contents of the **Functions** list on the **Coverage** tab to *filename*.
 - `coverage_source` — Exports the contents of the **Source Lines** list on the **Coverage** tab to *filename*. This exports source line information for all visible functions, not just the source lines visible in the **Profile** window.

- `blocks` — Exports the contents of the **Blocks** list on the **Blocks** tab to *filename*. This exports blocks for all visible functions, not just the blocks visible in the **Profile** window.

This option corresponds to **File** → **Export to CSV**.

- `display` — Controls the display of profile information in the Debugger window:
 - `none` — Does not display any profile information.
 - `percent` — Displays **Time %** or **Inst %** values for each line in the Debugger source pane. This option corresponds to the **Percentage View** button (.
 - `coverage` — Shades uncovered lines in the Debugger source pane and displays the number of times each line was executed, if this information is available. This option corresponds to the **Coverage View** button (.
- `clear` — Clears instrumented profiling data stored on the target.
- `close` — Closes the **Profile** window.

Chapter 13

Program Execution Command Reference

Contents

General Program Execution Commands	160
Continue Commands	162
Halt Commands	170
Run Commands	171
Signal Commands	176
Single-Stepping Commands	177
Synchronous Debugging Commands	185
Task Execution Commands	187

The commands in this chapter allow you to control the execution of programs in the Debugger.

General Program Execution Commands

The following list provides a brief description of each general program execution command. For a command's arguments and for more information, see the page referenced.

- **g** — Changes the program counter so that the specified address expression becomes the next instruction to be executed (see “g” on page 160).
- **getargs** — Shows the current arguments that will be passed to your program the next time it is run (see “getargs” on page 160).
- **setargs** — Sets arguments for the next time the program is started from MULTI (see “setargs” on page 161).

g

g *address_expression*

Changes the program counter so that *address_expression* becomes the next instruction to be executed. You cannot set the next execution point to an address that is outside the current function.

getargs

getargs

Shows the current arguments that will be passed to your program the next time it is run. Support for passing program arguments to applications started from MULTI is target and operating system specific.

The following example shows the use of **setargs**, **getargs**, and **r**. (See also “setargs” on page 161 and “r” on page 171.)

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

setargs

setargs [*program_arguments*]

Sets arguments for the next time the program is started from MULTI. Support for passing program arguments to applications started from MULTI is target and operating system specific. If no arguments are specified, no arguments will be passed to the program.

Arguments must be in a space-separated list and may contain <, >, or >> to redirect standard input (`stdin`) and standard output (`stdout`). Text between quotation marks, either single (' ') or double (" "), is treated as a single argument. The quotation marks are removed and are not sent to the program. Arguments containing the MULTI command syntax comment delimiter (//) must be enclosed in quotation marks (for example, `setargs -perform_url_operation "http://www.example.com"`).

On Windows, this command uses Windows command line argument parsing rules. This emulates the behavior of the Windows command prompt. For more information, see Microsoft's documentation about command line argument parsing. At the time of writing, this documentation was available at <http://msdn.microsoft.com/en-us/library/17w5ykft.aspx>.

On Linux/Solaris, a tilde (~) expands the same way as the shell if you are running `csh`. However, other shell processing, such as wildcard expansion and pipes, is not performed.

See also “getargs” on page 160 and “r” on page 171.

Corresponds to: **Debug** → **Set Program Arguments**

Continue Commands

The commands in this section allow you to continue a stopped process.

Some of the continue commands use the `@continue_count` argument to specify how many breakpoints the Debugger will pass before stopping. For example, if `continue_count` is 4, the Debugger will skip over the next three breakpoints and stop the process at the fourth breakpoint, unless stopped earlier by the optional address expression described below.



Note

Only breakpoints that stop program execution are counted. A conditional breakpoint whose condition is false or a breakpoint whose commands resume a process are not counted.

You can view the `continue_count` by using the `CONTINUECOUNT` system variable. See “System Variables” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

Most of the continue commands also accept an optional address expression. If specified, a temporary breakpoint is set at that location. Unless otherwise noted, the breakpoint is removed as soon as it is reached and the process is stopped, even if fewer than `continue_count` breakpoints were skipped. For more information about address expressions, see “Using Address Expressions in Debugger Commands” on page 5.



Note

In MULTI 4.x, if you wanted to debug your program from its start address (typically `_start`), you had to set a breakpoint at the start address and continue to it, even after using the **load** command to load your program. In MULTI 5.x and later, you can use **load** or **prepare_target -load** to debug your program from its start address. If you have already executed **load** or **prepare_target -load**, the continue commands will not hit a breakpoint set at the program start address. See “load” on page 251 and “prepare_target” on page 253.

The following list provides a brief description of each continue command. For a command's arguments and for more information, see the page referenced.

- **bc** — Runs a halted process backward (see “bc” on page 164).

- **bcb** — Runs a halted process backward, blocking further command processing until the target halts or until you interrupt this command by pressing **Esc** (see “**bcb**” on page 164).
- **bcbhardbrk** — Runs a halted process backward to a temporary hardware breakpoint, blocking further command processing until the target halts or until you interrupt this command by pressing **Esc** (see “**bcbhardbrk**” on page 165).
- **bchardbrk** — Runs a halted process backward to a temporary hardware breakpoint (see “**bchardbrk**” on page 165).
- **c** — Continues a halted process (see “**c**” on page 166).
- **cb** — Continues a halted process, blocking further command processing until the process has halted again or until you interrupt this command by pressing **Esc** (see “**cb**” on page 166).
- **cbhardbrk** — Continues a halted process to a temporary hardware breakpoint, blocking further command processing until the process has halted again or until you interrupt this command by pressing **Esc** (see “**cbhardbrk**” on page 167).
- **cf** — Continues a halted process from the given address expression instead of the current program counter (see “**cf**” on page 167).
- **cfb** — Continues a halted process from the given address expression instead of the current program counter, blocking further command processing until the process has halted again or until you interrupt this command by pressing **Esc** (see “**cfb**” on page 168).
- **chardbrk** — Continues a halted process to a temporary hardware breakpoint (see “**chardbrk**” on page 168).
- **runbacktohardbrk** — Runs a halted process backward to a temporary hardware breakpoint that is automatically removed the next time any task halts for any reason (see “**runbacktohardbrk**” on page 168).
- **runtohardbrk** — Continues a halted process to a temporary hardware breakpoint that is automatically removed the next time any task halts for any reason (see “**runtohardbrk**” on page 169).
- **runtohere** — Continues a halted process to the current line or address (see “**runtohere**” on page 169).

bc

bc [*@continue_count*] [*expr*]

TimeMachine/INTEGRITY TimeMachine command, GUI only

Runs a halted process backward to the previous breakpoint or, if no previous breakpoint exists, to the first instruction in the trace data (TimeMachine) or the log data (INTEGRITY TimeMachine).

Available options are:

- *@continue_count* — Specifies the continue count. For more information, see “Debugger Command Conventions” on page 3.
- *expr* — Specifies an address expression where a temporary breakpoint is set. If the temporary breakpoint is hit, the process stops even if fewer than *continue_count* breakpoints were skipped.

To interrupt this command, press **Esc**.

Corresponds to: 

bcb

bcb [*@continue_count*] [*expr*]

TimeMachine/INTEGRITY TimeMachine command, GUI only

Behaves like the **bc** command, except that no further commands will be processed until the target halts or you interrupt this command with **Esc** (see “bc” on page 164). For a discussion of command blocking, see “cb” on page 166.

bcbhardbrk

bcbhardbrk [read] [write] [execute] [mask=*mask*] [data=*data_value*]
[dmask=*data_mask*] [vm] [at] [--] *expr* [:*size*] [{*commands*}]

TimeMachine/INTEGRITY TimeMachine command, GUI only

Sets a temporary hardware breakpoint in TimeMachine/INTEGRITY TimeMachine mode, at the location specified by *expr*, and then executes the **bcb** command. The **bcb** command runs the halted process backward to the breakpoint, blocking further command processing until the target halts or until you interrupt the command by pressing **Esc** (see “bcb” on page 164).

An unlimited number of hardware breakpoints are supported in TimeMachine/INTEGRITY TimeMachine mode, regardless of target hardware support.

For argument descriptions, see “hardbrk” on page 59.

bchardbrk

bchardbrk [read] [write] [execute] [mask=*mask*] [data=*data_value*]
[dmask=*data_mask*] [vm] [at] [--] *expr* [:*size*] [{*commands*}]

TimeMachine/INTEGRITY TimeMachine command, GUI only

Sets a temporary hardware breakpoint in TimeMachine/INTEGRITY TimeMachine mode, at the location specified by *expr*, and then executes the **bc** command, which runs the halted process backward to the breakpoint (see “bc” on page 164).

An unlimited number of hardware breakpoints are supported in TimeMachine/INTEGRITY TimeMachine mode, regardless of target hardware support.

For argument descriptions, see “hardbrk” on page 59.

See also “runbacktohardbrk” on page 168.

c

c [*@continue_count*] [*expr*]

Continues a stopped process. Available options are:

- *@continue_count* — Specifies the continue count. For more information, see “Debugger Command Conventions” on page 3.
- *expr* — Specifies an address expression where a temporary breakpoint is set. If the temporary breakpoint is hit, the process stops even if fewer than *continue_count* breakpoints were skipped.

On Linux/Solaris, if the process stops because of a signal, the **c** command continues with or without the signal based on the current signal handling specified for that signal by the **signal** command (see “signal” on page 176).

To interrupt this command, press **Esc**.

See also “cu, cU” on page 180.

Corresponds to: 

Corresponds to: **Debug** → **Go on Selected Items**

cb

cb [*@continue_count*] [*expr*]

Behaves like the **c** command, except that no further commands will be processed until your process has stopped again. Signals for **cb** are handled as they are for **c**. See “c” on page 166.

Some commands, such as data printing and viewing commands, only work correctly when your process is stopped. When these commands appear in a script that controls your process, it is important to ensure that your process has stopped before executing these commands. Using the **cb** command makes sure your process stops running before further script commands are executed. Your process will stop running once it has done one of the following:

- Run to completion.

- Hit a breakpoint.
- Stopped with an exception, signal, segmentation violation, bus error, or similar cause.

To interrupt this command, press **Esc**.

cbhardbrk

cbhardbrk [read] [write] [execute] [mask=*mask*] [data=*data_value*]
[dmask=*data_mask*] [vm] [at] [--] *expr* [:*size*] [{*commands*}]

Sets a temporary hardware breakpoint at the location specified by *expr*, and then executes the **cb** command, which runs the halted process forward to the breakpoint, blocking further command processing until the process halts or until you interrupt the command by pressing **Esc** (see “cb” on page 166).

For argument descriptions, see “hardbrk” on page 59.

cf

cf *address_expression*

Continues a stopped process from the given *address_expression* instead of the current program counter. This will have the effect of skipping some of your program's code and is equivalent to issuing a **g** command followed by a **c** command (see “g” on page 160 and “c” on page 166). The given *address_expression* must describe a location within the currently executing function. See “Using Address Expressions in Debugger Commands” on page 5.

The following example installs a breakpoint on line 12 of function `foo`. When the process hits the breakpoint, it will continue from line 14 of function `foo`, effectively skipping lines 12 and 13 of function `foo`.

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

The following example installs a breakpoint at label `bar` of function `foo`. When the process hits the breakpoint, it will continue from the exit point of the function, effectively skipping the rest of the function and returning immediately.

```
> b foo##bar { cf ($retadr()) }
```

To interrupt this command, press **Esc**.

cfb

cfb *address_expression*

Behaves like the **cf** command, except that no further commands will be processed until your process has stopped again (see “cf” on page 167). For a discussion of command blocking, see “cb” on page 166.

To interrupt this command, press **Esc**.

chardbrk

chardbrk [read] [write] [execute] [mask=*mask*] [data=*data_value*]
[dmask=*data_mask*]] [vm] [at] [--] *expr* [:*size*] [{*commands*}]

Sets a temporary hardware breakpoint at the location specified by *expr*, and then executes the **c** command, which runs the halted process forward to the breakpoint (see “c” on page 166).

For argument descriptions, see “hardbrk” on page 59.

See also “runtohardbrk” on page 169.

runbacktohardbrk

runbacktohardbrk [read] [write] [execute] [mask=*mask*] [data=*data_value*]
[dmask=*data_mask*]] [vm] [at] [--] *expr* [:*size*] [{*commands*}]

TimeMachine/INTEGRITY TimeMachine command, GUI only

Sets a temporary hardware breakpoint in TimeMachine/INTEGRITY TimeMachine mode, at the location specified by *expr*, and then executes the **bc** command, which runs the halted process backward to the breakpoint. See “bc” on page 164.

The **runbacktohardbrk** command functions just like the **bchardbrk** command except that the temporary breakpoint set by **runbacktohardbrk** is automatically

removed the next time any task on the target halts, regardless of whether the breakpoint was hit, whereas the temporary breakpoint set by **bchardbrk** is automatically removed only when it is hit. See “bchardbrk” on page 165.

An unlimited number of hardware breakpoints are supported in TimeMachine/INTEGRITY TimeMachine mode, regardless of target hardware support.

For argument descriptions, see “hardbrk” on page 59.

runtohardbrk

runtohardbrk [read] [write] [execute] [mask=*mask*] [data=*data_value* [dmask=*data_mask*]] [vm] [at] [--] *expr* [:*size*] [{*commands*}]

Sets a temporary hardware breakpoint at the location specified by *expr*, and then executes the **c** command, which runs the halted process forward to the breakpoint. See “c” on page 166.

The **runtohardbrk** command functions just like the **chardbrk** command except that the temporary breakpoint set by **runtohardbrk** is automatically removed the next time any task on the target halts, regardless of whether the breakpoint was hit, whereas the temporary breakpoint set by **chardbrk** is automatically removed only when it is hit. See “chardbrk” on page 168.

For argument descriptions, see “hardbrk” on page 59.

runtohere

runtohere

Runs to the current line or address.

This command sets a temporary breakpoint on the current line or current address, and then executes the **c** command (see “c” on page 166). The temporary breakpoint is automatically removed the next time any task on the target halts, regardless of whether the breakpoint was hit.

As an alternative to using this command, you can double-middle-click anywhere on a line to make the process run to that line (assuming that you have not configured a double-middle-click to perform a different function).

To interrupt this command, press **Esc**.

Halt Commands

The commands in this section allow you to halt the process being debugged.

The following list provides a brief description of each halt command. For a command's arguments and for more information, see the page referenced.

- **H** — Prints the cause of a halt (see “H” on page 170).
- **halt** — Halts the current process (see “halt” on page 170).
- **k** — Kills the current process (see “k” on page 171).

H

H

Prints the cause of a halt.

halt

halt [*{commands}*]

Halts the current process. The process halts without sending an interrupt, allowing you to cleanly continue the process later.

If Debugger commands are specified in *commands* (see “Using Command Lists in Debugger Commands” on page 12), the specified commands will be executed when the process halts.

Corresponds to: 

Corresponds to: **Debug** → **Halt on Selected Items**

k

k [*force*]

Kills the current process. Specifying *force* kills the process without prompting you first.

Corresponds to: **Debug** → **Kill Selected Items**

Run Commands

The commands in this section allow you to run the program being debugged.

The following list provides a brief description of each run command. For a command's arguments and for more information, see the page referenced.

- **r** — Runs a new target program and passes the specified arguments to the program in a space-separated list (see “r” on page 171).
- **R** — Runs a new target process with no arguments (see “R” on page 172).
- **rb, Rb** — Runs or restarts the program (see “rb, Rb” on page 173).
- **restart** — Restarts program execution or resets aspects of program and target (see “restart” on page 173).
- **resume** — Resumes program execution after all other commands in a breakpoint command list have been issued (see “resume” on page 174).
- **rundir** — Changes the directory or prints the current run directory (see “rundir” on page 174).
- **runtask** — Starts a task running on a VxWorks target (see “runtask” on page 175).

r

r [*arguments*]

Runs a new target program and passes *arguments* to the program in a space-separated list. For example, the command `r fly 3`, runs the program with the two arguments `fly` and `3`. If a process already exists, the Debugger kills it and

then prepares the target using the current prepare target settings (see “prepare_target” on page 253).

If no arguments are specified, the ones from the previous run are used. If no previous run exists, no arguments are used.

Support for passing program arguments to applications started from MULTI is target and operating system specific.

Arguments may contain `<`, `>`, or `>>` to redirect standard input (`stdin`) and standard output (`stdout`). Text between quotation marks, either single (`' '`) or double (`"`), is treated as a single argument. The quotation marks are removed and are not sent to the program. Arguments containing the MULTI command syntax comment delimiter (`//`) must be enclosed in quotation marks (for example, `r -perform_url_operation "http://www.example.com"`).

On Windows, this command uses Windows command line argument parsing rules. This emulates the behavior of the Windows command prompt. For more information, see Microsoft's documentation about command line argument parsing. At the time of writing, this documentation was available at <http://msdn.microsoft.com/en-us/library/17w5ykft.aspx>.

On Linux/Solaris, a tilde (`~`) expands the same way as if you had entered it into the `csh` shell. However, other shell processing, such as wildcard expansion and pipes, is not performed.

This command is not supported for ROM programs.

See also “setargs” on page 161, “restart” on page 173, and “R” on page 172.

R

R

Runs a new target process with no arguments. If the process already exists, it will be killed and restarted.

This command is not supported for ROM programs.

See also “r” on page 171.

rb, Rb

rb [*arguments*]

Rb

Runs or restarts the program. These commands behave like the **r** and **R** commands except that no further commands will be processed until the process terminates, hits a breakpoint, or stops in any other way (see “r” on page 171 and “R” on page 172). While the command line input is blocked, you can still perform all interactive operations appropriate to a process, such as pressing the **Halt** button.

This command is useful for writing scripts that control execution of a process running on the target, since you often want to perform the next command only after the process stops.

The **Rb** command behaves like **rb**, except that it runs the program without arguments.

These commands are not supported for ROM programs.

To interrupt these commands, press **Esc**.

See also “cb” on page 166, “r” on page 171, and “R” on page 172.

restart

restart

Restarts program execution or resets aspects of program and target, depending on debugging context and specified arguments.

- During native and embedded debugging, this command is identical to the **r** command with no arguments (see “r” on page 171).
- During debugging of Dynamic Download INTEGRITY applications, this command attempts to (re)load the application. This command may not be available for use with relocatable modules.

This command is not supported for ROM programs.

Corresponds to: 

Corresponds to: **Debug** → **Restart**

resume

resume [*address_expression*]

(This command is only valid within a breakpoint command list. See “Using Command Lists in Debugger Commands” on page 12.)

Resumes program execution at the specified *address_expression* after all other commands in the breakpoint command list have been issued. If no *address_expression* is specified, the process will resume from the location of the breakpoint. See “Using Address Expressions in Debugger Commands” on page 5.

For example, to skip over line 5 in your program, you could use the following command, which makes the process stop at line 5 and then resume execution at line 6:

```
> b 5 {resume 6}
```

resume will continue the process in the same manner that the breakpoint was encountered. For example, if the Debugger was performing a **c** (continue) command when the breakpoint was encountered, the **c** command will be resumed. If the Debugger was performing an **S** (single-step) command and the step completed when the breakpoint was encountered, the process stops.

rundir

rundir [*dir* | -clear]

Changes the directory or prints the current run directory, where:

- *dir* — Changes the directory in which the process runs to *dir*. The run directory setting is saved between sessions.

For embedded processes that use host I/O, *dir* becomes the directory that MULTI uses to perform host I/O operations. Processes that have already started running when the **rundir** command is issued are not affected by the new host I/O directory setting.

For information about the GUI equivalent of the **rundir *dir*** command, see the description of the **Start in** field in “The Arguments Dialog Box” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book.

- **-clear** — Removes any saved run directory setting and changes the run directory back to the default directory. The default directory is MULTI's current working directory, which you can display with the **pwd** command. See “pwd” on page 120.

If you do not specify an argument, this command prints the current run directory.

runtask

runtask *func* [*args*]

Starts a task running on a VxWorks target, where:

- *func* is the name of any downloaded function.
- *args* is a list of space delimited arguments to pass to the function. Acceptable values for *args* are:
 - Decimal and hexadecimal numeric constants.
 - Character constants.
 - String constants enclosed in double quotation marks (" ").
 - Names of global variables (the & operand cannot be used here).
 - I/O redirection operators < and >.

During C++ debugging, *func* may be the member function of a global object, specified as *object.function*. If the requested function is ambiguous, MULTI will open a dialog box showing all the options so you can choose the correct one.

Signal Commands

The commands in this section are only applicable to Linux/Solaris targets.

The following list provides a brief description of each signal command. For a command's arguments and for more information, see the page referenced.

- **signal** — Sends the given signal to the specified process or to the current process (see “signal” on page 176).
- **zsignal** — Sets up the signal handling table (see “zsignal” on page 176).

signal

signal *signal* [*pr=num*]

Linux/Solaris targets only

Sends the signal *signal* to the process specified by slot number *num*, or to the current process if *num* is not specified.



Note

Sending a fatal signal (for example, SIGKILL) to a stopped process may have unpredictable results.

zsignal

zsignal *signal* [*s*] [*i*] [*r*] [*b*] [*C*] [*Q*]

Linux/Solaris targets only

Sets up the signal handling table. To list the current signal settings, use the **l z** command (see “l” on page 115).

The optional flags are described below.

- *s* — Toggles *stop*. If *stop* is on, the process stops when the signal occurs.
- *i* — Toggles *ignore*. If *ignore* is on, the Debugger does not send the signal to the process.

- `r` — Toggles `report`. If `report` is on, a message is displayed every time the signal occurs.
- `b` — Toggles `bell`. If `bell` is on, 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.

For example, if the default state is do not stop, do not ignore, do not report, and no bell, the command `signal 14 sr` sets the alarm clock signal to stop, do not ignore, report, and no bell. Running `signal 14 sr` again toggles these flags back to the previous state. Running `signal 14 Csb`, in any signal state, will set the alarm clock signal to stop, do not ignore, beep, and do not report.

Modifying the state of the “breakpoint” signal (usually SIGTRAP) is not supported.

Single-Stepping Commands

The commands in this section allow you to single-step through your program. The commands differ in whether they allow you to step into or step over function calls and whether they advance by a single machine instruction or a single source line, as shown in the table below.

	Steps into function calls	Steps over function calls
Advances one source line	sl	Sl nl
Advances one machine instruction	si	Si ni
Advances one machine instruction when in either assembly-only mode or interlaced assembly mode. Advances one source line in source-only mode.	s	S n

The single-stepping commands listed in the preceding table accept the following optional parameters:

- *num* — Specifies how many single-steps to perform. If no *num* is specified, one step is performed. If a breakpoint is encountered before *num* steps have taken place, the remainder of the steps are aborted.
- *n* | *b* — Specifies whether Debugger commands are blocked during the single-step. If *b* (blocking) is specified, no Debugger commands will be executed until the step finishes. If *n* is specified, subsequent Debugger commands can execute before the step is finished. If neither *n* nor *b* is specified, the step will be blocking or non-blocking according to the `BlockStep` configuration option, which defaults to non-blocking, but can be changed from the configuration GUI or via the **configure BlockStep** command. For more information about the `BlockStep` configuration option, see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.



Tip

If you inadvertently step into a function (with an **s**, **si**, or **sl** command), you can issue the **cU** command (see “cu, cU” on page 180), click the  button, or press **F9** to return from the function.



Note

The Debugger steps by source line without regard to asynchronous changes to control flow that may occur in your program, such as those caused by signals (in the C/POSIX environment) or Interrupt Activities (in the INTEGRITY environment). If your step is stopped by a signal or Interrupt Activity, you can issue the **cU** command, click the  button, or press **F9** as necessary to return from the invocation of the signal handler or Interrupt Activity handler.

The following list provides a brief description of each single-stepping command. For a command's arguments and for more information, see the page referenced.

- **bcU** — Steps backward, up to the caller of the current function (see “bcU” on page 179).
- **bprev** — Steps backward one source line, stepping over function calls (see “bprev” on page 180).

- **bs** — Steps backward one source line (or one machine instruction in assembly-only mode or interlaced assembly mode), stepping into function calls (see “bs” on page 180).
- **bsi** — Steps backward one machine instruction (see “bsi” on page 180).
- **cu, cU** — Steps up to the caller of the current function or to the specified address expression (see “cu, cU” on page 180).
- **s** — Single-steps one source line (or one machine instruction in assembly-only mode or interlaced assembly mode), stepping into any function calls (see “s” on page 181).
- **S, n** — Single-steps one source line (or one machine instruction in assembly-only mode or interlaced assembly mode), stepping over function calls (see “S, n” on page 182).
- **si** — Single-steps one machine instruction, stepping into function calls (see “si” on page 182).
- **Si, ni** — Single-steps one machine instruction, stepping over function calls (see “Si, ni” on page 183).
- **sl** — Single-steps one source line, stepping into function calls (see “sl” on page 183).
- **Sl, nl** — Single-steps one source line, stepping over function calls (see “Sl, nl” on page 184).
- **stepinto** — Sets a temporary breakpoint in the supplied function and steps once (see “stepinto” on page 184).

bcU

bcU

TimeMachine/INTEGRITY TimeMachine command, GUI only

Steps backward, up to the caller of the current function. This command stops on the call instruction that called the current function; this may be in the middle of a source line. This behavior mirrors that of the **cU** command (see “cu, cU” on page 180).

Corresponds to: 

bprev

bprev

TimeMachine/INTEGRITY TimeMachine command, GUI only

Steps backward one source line, stepping over function calls.

Corresponds to: 

bs

bs

TimeMachine/INTEGRITY TimeMachine command, GUI only

Steps backward one source line, stepping into function calls. In assembly-only mode or interlaced assembly mode, this command will step one machine instruction instead of one source line. To step backward one machine instruction unconditionally, use the **bsi** command instead (see “bsi” on page 180).

Corresponds to: 

bsi

bsi

TimeMachine/INTEGRITY TimeMachine command, GUI only

Steps backward one machine instruction. For source-level stepping, use the **bs** command (see “bs” on page 180).

cu, cU

cu [*address_expression*]

cU [*address_expression*]

With no argument, steps up to the caller of the current function. This command stops on the instruction that the current function returns to; this may be in the middle

of a source line. This command is useful if you have accidentally single-stepped into a function that you meant to step over, or if you want execution to proceed to another place further up the call stack.

If *address_expression* is specified, this command steps up to the caller of the current function or to the temporary breakpoint set at *address_expression*—whichever is reached first. For more information about address expressions, see “Using Address Expressions in Debugger Commands” on page 5.

The **cu** command sets a permanent breakpoint at the return site of the currently executing function. The **cU** command sets a temporary breakpoint at the return site of the currently executing function. The **cu** and **cU** commands handle signals like the **c** command. See “c” on page 166.

The **cu** and **cU** commands rely on the Debugger's ability to generate a partial stack trace. They may not work correctly (for example, they may set a breakpoint at the wrong address) if the stack trace obtained by the Debugger is incorrect. For restrictions on tracing the call stack, see “Viewing Call Stacks” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

For information about continuing a stopped process from an up-level breakpoint, see “bu, bU” on page 53.

cU corresponds to: 

cU corresponds to: **Debug** → **Return on Selected Items**

S

s [*num*] [*n* | *b*]

Single-steps one source line, stepping into any function calls. The options *num*, *n*, and *b* behave as specified in “Single-Stepping Commands” on page 177.

In assembly-only mode or interlaced assembly mode, the **s** command steps one machine instruction instead of one source line. For information about these modes, see “Source Pane Display Modes” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

To interrupt this command, press **Esc**.

Corresponds to: **F11**

Corresponds to: 

Corresponds to: **Debug → Step (into Functions) on Selected Items**

S, n

S [*num*] [*n* | *b*]

n [*num*] [*n* | *b*]

Single-steps one source line, stepping over function calls instead of into functions. The options *num*, *n*, and *b* behave as specified in “Single-Stepping Commands” on page 177.

In assembly-only mode or interlaced assembly mode, these commands step one machine instruction instead of one source line. For information about these modes, see “Source Pane Display Modes” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

To interrupt these commands, press **Esc**.

Corresponds to: **F10**

Corresponds to: 

Corresponds to: **Debug → Next (over Functions) on Selected Items**

si

si [*num*] [*n* | *b*]

Single-steps one machine instruction, stepping into function calls. The options *num*, *n*, and *b* behave as specified in “Single-Stepping Commands” on page 177.

This command behaves like the **s** command (see “s” on page 181), except that the **si** command causes the process to advance by one machine instruction instead of

one high-level source line. Furthermore, the stop position is displayed as a disassembled instruction.

To interrupt this command, press **Esc**.

Si, ni

Si [*num*] [*n* | *b*]

ni [*num*] [*n* | *b*]

Single-steps one machine instruction, stepping over function calls. The options *num*, *n*, and *b* behave as specified in “Single-Stepping Commands” on page 177.

These commands behave like the **S** and **n** commands (see “S, n” on page 182), except that the **Si** and **ni** commands cause the process to advance by one machine instruction instead of one high-level source line, and the stop position is displayed as a disassembled instruction.

To interrupt these commands, press **Esc**.

sl

sl [*num*] [*n* | *b*]

(This command is a lowercase **s** and a lowercase **l**.)

Single-steps one source line (even if you are viewing your code in interlaced assembly mode), stepping into function calls. The options *num*, *n*, and *b* behave as specified in “Single-Stepping Commands” on page 177.

To interrupt this command, press **Esc**.

SI, nl

SI [*num*] [*n* | *b*]

(This command is an uppercase **S** and a lowercase **L**.)

nl [*num*] [*n* | *b*]

(This command is a lowercase **N** and a lowercase **L**.)

Single-steps one source line (even if you are viewing your code in interlaced assembly mode), stepping over function calls. The options *num*, *n*, and *b* behave as specified in “Single-Stepping Commands” on page 177.

To interrupt these commands, press **Esc**.

stepinto

stepinto *expr*

Sets a temporary breakpoint in the supplied function and steps once. This command is useful in situations where there are a number of functions on a single line but you are interested in stepping into only one of them. The breakpoint used is a special type of breakpoint that will only trigger one stack level below the current stack level. This command performs the same action as the **Step Into This Function** right-click menu option.

Synchronous Debugging Commands

The command in the following section allows you to synchronously pause or unpause tasks in your INTEGRITY run-mode system.

syncrc

syncrc (enable | on) | (disable | off) | (toggle | tog)

syncrc pause | unpause | togpause

syncrc (exclude | include | inherit) (-current_task | -tid *ID* | -taskname *name* [-current_as | -aid *ID* | -asname *name*]) | (-current_as | -aid *ID* | -asname *name*)

The above parentheses are used for grouping; they are not a part of the command.

The first format of the **syncrc** command enables or disables synchronous debugging, the second pauses or unpauses tasks in the system, and the third allows you to control which tasks can be paused. For more information, see “Stopping and Starting Tasks Synchronously” in Chapter 8, “Executing and Controlling Your Program from the Debugger” in *MULTI: Debugging*.

This command is only supported with run-mode connections to INTEGRITY IoT.

Available options are:

- **enable | on** — Enables synchronous debugging for all AddressSpaces and tasks in the system except for those that are explicitly excluded (see **exclude** below), loader tasks, and write-protected tasks. This option is the default until you disable synchronous debugging, at which point your last setting is remembered across MULTI sessions and applied to every subsequent run-mode connection to INTEGRITY IoT.
- **disable | off** — Disables synchronous debugging.
- **toggle | tog** — Toggles synchronous debugging on or off.
- **pause** — Pauses all AddressSpaces and tasks for which synchronous debugging has been enabled.
- **unpause** — Unpauses all AddressSpaces and tasks for which synchronous debugging has been enabled. Halted tasks are resumed going forwards (as when

you click ). To unpause the system and resume halted tasks going backwards, click  while in INTEGRITY TimeMachine mode.

- `togpause` — Toggles between pause and unpause.
- `(exclude|include|inherit) (-current_task|-tid ID|-taskname name [-current_as|-aid ID|-asname name]) | (-current_as|-aid ID|-asname name)` — Excludes the specified task or AddressSpace from synchronous debugging, includes it again, or specifies that the exclusion/inclusion status of the specified task or AddressSpace should be inherited from the parent AddressSpace or from the system, respectively. Exclusion/inclusion status is inherited by default.

This option applies only to the current MULTI session. Excluding a task is useful if pausing the task would have adverse real-world effects. Including a task is useful when it is the exception to an AddressSpace you have excluded.

`-current_task` and `-current_as` determine the task and AddressSpace based on the current target list selection.

If you specify a task name that is not unique, specify the AddressSpace that contains the task. In this case, only the task, and not the AddressSpace, is affected.

Excluding or including an AddressSpace excludes or includes all the current and future tasks within the AddressSpace that are set to inherit from it. This is true for the duration of the current MULTI session.

You may specify only one task or AddressSpace each time you issue the `syncrc exclude`, `syncrc include`, or `syncrc inherit` command.



Note

Excluding individual tasks or all the tasks in an INTEGRITY TimeMachine-instrumented AddressSpace prevents all future use of INTEGRITY TimeMachine and History with that AddressSpace. It may also cause problems for INTEGRITY TimeMachine use with other AddressSpaces.

Task Execution Commands

The command in the following section allows you to control one or more run-mode tasks.

taskaction

taskaction -r|-h|-s [-addressSpace *address_space_name*] [-taskname] *task_name1* [*task_name2*]... | [-taskid] *task_id1* [*task_id2*]...

Performs an operation on the run-mode task(s) specified by task name or task ID. Possible operations are:

- -r — Resumes the task(s).
- -h — Halts the task(s).
- -s — Single-steps the task(s).

If neither `-taskname` nor `-taskid` is specified, MULTI assumes that numeric entries are task IDs and that other entries are task names.

The `-addressSpace` option is used to specify an INTEGRITY AddressSpace in which the specified task or tasks exist. Use this option if you want to refer to a task by name, but more than one AddressSpace contains a task with that name. This option is only meaningful if you are debugging an INTEGRITY process.

Chapter 14

Register Command Reference

Contents

Register Commands	190
-------------------------	-----

The commands in this chapter allow you to modify register definitions while debugging a program; open windows for viewing registers; and add, remove, load, and save registers. For more information about working with registers, see Chapter 13, “Using the Register Explorer” in the *MULTI: Debugging* book.



Note

Any modifications to the register descriptions made from the command line are active only until you reload the program or connect to a different target. For persistent modifications, you must customize the default register description files. See “Customizing Default Register Definition Files” in Chapter 13, “Using the Register Explorer” in *MULTI: Debugging*.

Register Commands

The following list provides a brief description of each register command. For a command's arguments and for more information, see the page referenced.

- **regadd** — Dynamically adds a memory-mapped register into the Debugger context and opens a **Register Setup** dialog, which allows you to specify other basic information for the register (see “regadd” on page 191).
- **regappend** — Loads the register description file specified by the given file, and applies the modifications to the registers defined in the Debugger (see “regappend” on page 191).
- **regbasefile** — Prints out the full path to the file that is used as the base for the register descriptions of the active Debugger (see “regbasefile” on page 192).
- **regload** — Removes all of the registers that are currently defined in the Debugger and creates a new set of registers from the register definition file specified (see “regload” on page 192).
- **regtab** — Modifies the configuration of the specified tab on all open **Register View** windows (see “regtab” on page 192).
- **regunload** — Removes all of the registers that are defined in the specified file (see “regunload” on page 194).
- **regvalload** — Loads register values from the specified file (see “regvalload” on page 194).
- **regvalsave** — Saves register values (see “regvalsave” on page 194).

- **regview** — Opens a **Register View** window displaying all registers, or opens a **Register Information** window displaying the specified register (see “regview” on page 195).

regadd

regadd *name address [size_in_bytes [memory_space]]*

Dynamically adds a memory-mapped register in the current context and opens the **Register Setup** dialog box, which allows you to specify additional information about the register. See “The Register Setup Dialog” in Chapter 13, “Using the Register Explorer” in the *MULTI: Debugging* book.

Available arguments are:

- *name* — Specifies a unique name for the register.
- *address* — Specifies the memory address of the register.
- *size_in_bytes* — Specifies the memory size of the register.
- *memory_space* — Specifies the memory space of the register. For more information, see the description of *memory_space* in “The general Section” in Appendix F, “Register Definition and Configuration Reference” in *MULTI: Debugging*.

regappend

regappend *filename*

Loads the register description file specified by *filename*, and applies the modifications to the registers defined in the Debugger. If you do not provide an absolute path to *filename*, MULTI searches the following directories in the order listed:

1. The current working directory.
2. The **registers** directory located in your personal configuration directory.
3. The **registers** directory located in the site-wide configuration directory.
4. The **registers** directory located in the MULTI **defaults** directory.

By storing frequently used register additions and modifications in a file, you can use **regappend** to quickly insert them into your Debugger.

regbasefile

regbasefile

Prints out the full path to the file that is used as the base for the register descriptions of the active Debugger. This is the full path to the root `GRD` file—the file that was automatically loaded by **MULTI** or the file from the most recent **regload** command (see “regload” on page 192).

regload

regload [*filename*]

GUI only

Removes all of the registers that are currently defined in the Debugger and creates a new set of registers from the register definition file *filename*. If you do not provide an absolute path to *filename*, **MULTI** performs an ordered search of the directories listed in “regappend” on page 191. If no filename is specified, the default register definition files are reloaded.

We recommend that you use the **regappend** command (see “regappend” on page 191) unless the file you are loading contains one of the default register description files via a `%include` directive, or describes all of the registers you want to use.

regtab

regtab [-norefresh] *operation tab*

GUI only

Modifies the configuration of the specified tab on all open **Register View** windows, where:

- `-norefresh` — Specifies that open **Register View** windows will not be updated to reflect the changes indicated by the **regtab** command. This option

is useful for avoiding screen flicker when issuing multiple consecutive **regtab** commands in scripts or loops on the command line.

- *operation* — Specifies the configuration modification to be performed on the tab specified by *tab*. One of the operations listed below must be specified.
 - `-show group` — Shows the group or register identified by the dot-separated path *group*.
 - `-hide group` — Hides the group or register identified by the dot-separated path *group*.
 - `-reroot group` — Re-roots the tab at the group specified by the dot-separated path *group*.
 - `-unroot` — Unroots the tab.
 - `-insert` — Adds a new tab named *tab*.
 - `-remove` — Deletes the specified tab.
 - `-promote` — Moves the tab one to the left in the tab ordering.
 - `-demote` — Moves the tab one to the right in the tab ordering.
 - `-activate` — Makes the specified tab the active tab.

The *group* argument required by some of the above options indicates the group or register to use in the operation, and is a path that lists the name of the item to be used, preceded by the names of all of its parent groups. For example, to specify a register named `f0` that is contained within a 64-bit subgroup of the Floating Point Registers group, the dot-separated group path would be:

```
Floating Point Registers.64-bit.f0
```

Quotation marks are not required around the group path even if the path contains spaces.

- *tab* — Specifies the tab to be modified and must be the last argument for every **regtab** command. The tab name must contain only alphanumeric characters and underscores and does not require quotation marks.

This command only changes the appearance of open **Register View** windows. It has no effect if no **Register View** windows are open in the current Debugger.

regunload

regunload *filename*

Removes all of the registers that are defined in the file *filename*, which has been incrementally loaded using the **regappend** command (see “regappend” on page 191) or the **File** → **Load Register Definitions from File** menu option in the **Register View** window.

regvalload

regvalload [*filename*]

Loads register values from *filename*. If no filename is specified, a file chooser for register value files (**.grv**) is displayed so you can select a file containing register values.

For information about the simple syntax of register value files, examine a saved register file (see “regvalsave” on page 194).

regvalsave

regvalsave [-all|-tab|-selected] [-nomemmapped] [*filename*]

Saves register values, where:

- **-all** — Saves values for all registers in the Debugger. This is the default behavior.
- **-tab** — Saves values for all registers visible in the current tab of the **Register View** window. (See “The Register View Window” in Chapter 13, “Using the Register Explorer” in the *MULTI: Debugging* book.)
- **-selected** — Saves values for all selected registers in the current tab of the **Register View** window.
- **-nomemmapped** — Prevents memory-mapped registers from being saved. This option qualifies the behavior of **-all**, **-tab**, and **-selected**.
- *filename* — Specifies the **.grv** file in which to save the register values. If no filename is specified, a file chooser prompts you to select a **.grv** file.

regview

regview [*register_name*]

GUI only

Opens a **Register View** window displaying all of the registers, or opens a **Register Information** window displaying the specified register. The leading \$ in register name will be omitted. See “The Register View Window” in Chapter 13, “Using the Register Explorer” in the *MULTI: Debugging* book.

Corresponds to: 

Corresponds to: **View** → **Registers**

Chapter 15

Scripting Command Reference

Contents

Command Manipulation and Debugger Macro Commands	198
Conditional Program Execution Commands	204
Dialog Commands	208
External Tool Commands	211
History Commands	214
Hook Commands	217
MULTI-Python Script Commands	221
Object Structure Awareness (OSA) Commands	223
Record and Playback Commands	230

The commands in this chapter are particularly useful within scripts. See also Chapter 1, “Using MULTI Scripts” in the *MULTI: Scripting* book.

Command Manipulation and Debugger Macro Commands

The commands in this section manipulate other commands or deal with Debugger macros.

The following list provides a brief description of each command manipulation and Debugger macro command. For a command's arguments and for more information, see the page referenced.

- **alias** — Creates or lists aliases (see “alias” on page 199).
- **cedit** — Prints the output of the specified command to an Editor window (see “cedit” on page 199).
- **define** — Creates a Debugger macro for later use (see “define” on page 200).
- **macrotrace** — Prints the stack of all presently executing Debugger macro commands (see “macrotrace” on page 200).
- **return** — Returns from the currently executing Debugger macro, evaluating the specified expression and returning it as the macro value (see “return” on page 201).
- **route** — Routes the specified command to the specified component (see “route” on page 201).
- **sc** — Performs syntax checking on either a single command or an entire script file and all nested script files (see “sc” on page 201).
- **shell** — Invokes a shell to run the specified shell commands (see “shell” on page 202).
- **substitute** — Executes the output of the specified command string as a Debugger command (see “substitute” on page 203).
- **unalias** — Reverses a previous **alias** command (see “unalias” on page 204).

alias

alias [*string1* [*string2*]]

Creates or lists aliases.

- If no strings are specified as arguments, the **alias** command lists all aliases.
- If only one *string* argument is specified, the **alias** command lists the alias (that is, the value that is substituted for the *string* by the alias), if any exists.
- If two strings are specified, the **alias** command translates *string1*, when encountered as the first word in a command, into *string2*. Substitution is only performed once, so references to other aliases are ignored. For example, if you enter:

```
> alias sh showdef
```

you will be able to type `sh` instead of `showdef` to use the **showdef** command.

string1 must follow the rules of C/C++ identifiers. It may not be quoted, may not contain spaces, and must begin with a letter or underscore.

See also “`unalias`” on page 204.

cedit

cedit *command*

GUI only

Executes the *command* specified as its argument and places the command output in an Editor window. This is useful for examining the output of commands that print large amounts of information.

define

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

Creates a Debugger macro for later use.

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

The *body* of the macro is a command list that can contain **if** statements and loops. See “Using Command Lists in Debugger Commands” on page 12, and see the descriptions of the **if**, **do**, **for**, and **while** commands in “Conditional Program Execution Commands” on page 204.

Debugger macros can also return a value by using the **return** command in the *body*. See “return” on page 201.

For more information about this command, see “Debugger Macros” in Chapter 14, “Using Expressions, Variables, and Function Calls” in *MULTI: Debugging*.

macrotrace

macrotrace

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

```
> define mac1() {return mac2();}  
> define mac2() {return mac3();}  
> define mac3() {macrotrace; return 42;}
```

the following would be displayed if you enter `mac1()`:

```
> mac1()  
0 mac3()  
1 mac2()  
2 mac1()  
42
```

See also “define” on page 200.

return

return [*expr*]

Returns from the currently executing Debugger macro, evaluating *expr*, if specified, and returning it as the macro value.

This command is only valid in Debugger macros (see “define” on page 200). When a Debugger macro is running and the **return** command is issued, the macro stops and exits. If an expression *expr* is specified, it is evaluated and returned as the macro's value.

route

route *destination_component command*

Routes the specified command to the specified destination component. Note that *destination_component* overrides the selection in the target list, causing *command* to always execute on *destination_component* and never on the currently selected target list entry.

The component may be fully specified, or just the unique portion of the component name may be used. For example, `debugger.pid.543` and `pid.543` are equivalent as long as the latter is unique. If a match is not unique, a list of matching components is printed out.

To list components and their aliases, use the **components** command (see “components” on page 110).

To create a new alias for a new component, use the **new** command (see “new” on page 31).

sc

sc ["*command*" | <*filename*]

Performs syntax checking on either a single command or an entire script file and all nested script files. See “Syntax Checking” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

The filename will be searched for using the default search path. See “Default Search Path for Files Specified in Commands” on page 14.

shell

shell [*-wait* | *-w*] [*-noconsole*] *commands*

Invokes a shell to run the specified shell *commands*, where:

- *-wait* | *-w* — Causes MULTI to wait for shell commands to finish before continuing. This option is only applicable in GUI mode. In non-GUI mode, MULTI always waits for shell commands to finish. You can abort the waiting process by pressing **Esc**.
- *-noconsole* — Prevents MULTI from creating a new console window in which to run the shell commands (but does not prevent commands from opening their own GUIs, etc.). If you do not specify this option, you can see shell command output and/or type input to the commands from the console window. This option is only applicable on Windows.

Before being passed to the shell, the command string following **shell** is processed and all instances of the escape sequence `%EVAL{multi_commands}` are replaced by the result of evaluating *multi_commands*. This is useful for constructing dynamic arguments (that is, arguments that vary depending on your current debugging context) to shell tools. For example, to run a tool on the current file, construct a command of the form:

```
> shell toolname constant_args %EVAL{$_FILE}
```

Use the **shellConfirm** configuration option to govern behavior of the shell or command window used by *commands* (see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book).

substitute

substitute *cmd_string*

Executes the output of *cmd_string* as a Debugger command.

The *cmd_string* argument contains the template of the command to be executed. Within *cmd_string*, you may use the escape sequence `%EVAL{commands}` to evaluate expressions or Debugger commands within the command list *commands* and to perform substitutions in *cmd_string*. The output that the Debugger would print if *commands* were executed directly is instead captured and inserted as plain text into *cmd_string* in place of the `%EVAL` sequence. The output of a `%EVAL` sequence is substituted directly, and includes any newline characters or other output formatting provided by the Debugger, except that if the output returned by `%EVAL` for the evaluated expression or Debugger commands begins and ends with double quotation marks (`"`), these quotation marks will be removed. You may need to use the **mprintf** command to properly format complicated output (see “mprintf” on page 117).

You can use more than one `%EVAL` sequence. After all sequences have been replaced with the output of the respective expressions or commands, the Debugger executes the resulting *cmd_string*.

As an example use of this command, suppose you want to use a graphical file chooser to specify the path to a file you are going to edit. You could enter:

```
> substitute edit "%EVAL{filedialog}"
```

In the above example, the Debugger command **filedialog** returns a chosen file path, but it will not be quoted. In order to handle the case where the chosen file path contains spaces, a pair of quotation marks is placed around the `%EVAL` sequence in the example.



Note

The **implicitEvalEcho** configuration option may have an effect on the expected output of this command. If this option is set to `off`, the values of expressions are not echoed, preventing expressions contained in *commands* from being evaluated or subsequently substituted in *cmd_string*. In this case, be sure to use the **mprintf** command to force

the values of expressions specified in *commands* to echo. For example, rather than using a command like the following:

```
> substitute memread 4 %EVAL{$addr}
```

you should use this command:

```
> substitute memread 4 %EVAL{mprintf("0x%x", $addr)}
```

For more information, see “mprintf” on page 117 and the **implicitEvalEcho** option in “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

unalias

unalias *string*

Reverses a previous **alias** command (disassociates *string* from its substitution). For example, if **sh** is aliased to **showdef**, the command:

```
> unalias sh
```

disassociates **sh** from the **showdef** command.

See also “alias” on page 199.

Conditional Program Execution Commands

The commands in this section allow you to set conditions that must be met before commands are executed.

Composite commands for conditional program execution follow the same syntax rules as all MULTI Debugger commands. For information that may be helpful when entering conditional program execution commands, see “Using Command Lists in Debugger Commands” on page 12, “Continuing Commands onto Subsequent Lines” on page 13, and “Terminating Commands” on page 14.

The following list provides a brief description of each conditional program execution command. For a command's arguments and for more information, see the page referenced.

- **break** — Breaks out of a loop created with the Debugger **do**, **for**, or **while** command (see “break” on page 205).
- **continue** — Causes the current iteration of a loop created with the Debugger **do**, **for**, or **while** command to terminate and the next iteration to begin (see “continue” on page 205).
- **do** — Executes the specified command list at least once, and then for as long as the specified expression evaluates to a nonzero value (see “do” on page 206).
- **for** — Executes *init-expr* once, then executes the specified command list and the increment *inc-expr* for as long as the specified expression evaluates to a nonzero value (see “for” on page 206).
- **if** — Specifies conditional command execution (see “if” on page 207).
- **while** — Executes the specified command list for as long as the specified expression evaluates to a nonzero value (see “while” on page 208).

break

break [-fail | -succeed]

Breaks out of a loop created with the Debugger **do**, **for**, or **while** command, where:

- **-fail** — Causes **break** to be treated as a failure. This can be used to abort downloading to a target from a setup script if the script detects a failure condition.
- **-succeed** — [default] Causes **break** to be treated as a successful action.

This is similar to the **break** statement in C.

See also “do” on page 206, “for” on page 206, and “while” on page 208.

continue

continue

Causes the current iteration of a loop created with the Debugger **do**, **for**, or **while** command to terminate and the next iteration to begin. For the **do** and **while** commands, this means the condition is tested; for the **for** command, the increment

is executed. This is similar to the **continue** statement in C. See “do” on page 206, “for” on page 206, and “while” on page 208.

do

do {*commands*} while (*expr*)

Executes the command list *commands* at least once, and then as long as *expr* (an expression in the current language) evaluates to a nonzero value. For example:

```
> do {  
continued>     mprintf("%d\n", $i);  
continued>     $i++;  
continued> } while ($i<20)
```

In this case, the value of (*\$i*) will always be printed at least once, regardless of its initial value. This is similar to the **do** statement in C. See “Using Command Lists in Debugger Commands” on page 12.

To interrupt this command, press **Esc**.

for

for ([*init-expr*] ; [*cond*] ; [*inc-expr*]) {*commands*}

Executes *init-expr* once, then executes the command list *commands* and the increment *inc-expr* as long as *cond* (an expression in the current language) evaluates to a nonzero value.

For example:

```
> for ($i=0; $i<20; ++$i) {  
continued>     if($i%2==0) {  
continued>         print "even";  
continued>     } else {  
continued>         print "odd";  
continued>     }  
continued> }
```

In this case, the variable (`$i`) is initialized to 0, and the body of the loop is then executed twenty times. See “Using Command Lists in Debugger Commands” on page 12.

Any of *init-expr*, *cond*, or *inc-expr* may be empty. If *init-expr* or *inc-expr* is empty, there will be, respectively, no initialization or increment executed. If *cond* is empty, it is taken to be the value 1 and the loop will continue to execute until halted. This is similar to the **for** statement in C.

To interrupt this command, press **Esc**.

if

if (*expr*) { *commands* } [else if (*expr*) { *commands* }] ... [else { *commands* }]

Specifies conditional command execution. If the first expression *expr* evaluates to a nonzero value, the first group of specified commands (see “Using Command Lists in Debugger Commands” on page 12) is executed. However, if the first expression evaluates to zero and there are subsequent **else if** clauses, the commands in the first **else if** clause with a nonzero expression are executed. If there is an **else** clause, and the **if** clause and any **else if** clauses resolve to zero, the commands in the **else** clause are executed. This is similar to the **if** statement in C.

This command can be nested.

The following example sets a breakpoint that conditionally prints information:

```
> b main {
continued>   if(argc==1) {
continued>     print "one";
continued>   } else if (argc==2) {
continued>     print "two";
continued>   } else {
continued>     print "many";
continued>   }
continued> }
```

while

while (*expr*) {*commands*}

Executes the command list *commands* as long as *expr* (an expression in the current language) evaluates to a nonzero value. For example:

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

In this case, if (*\$j*>50) is true, the loop will terminate regardless of the value of (*\$i*). This is similar to the **while** statement in C. See “Using Command Lists in Debugger Commands” on page 12.

To interrupt this command, press **Esc**.

Dialog Commands

The commands in this section allow you to open dialog boxes.

The following list provides a brief description of each dialog command. For a command's arguments and for more information, see the page referenced.

- **alertdialog** — Displays a dialog box containing the specified string (see “alertdialog” on page 209).
- **dialog** — Opens the predefined dialog box specified (see “dialog” on page 209).
- **directorydialog** — Opens a directory chooser and returns the name of the directory that is selected from the chooser (see “directorydialog” on page 210).
- **filedialog** — Opens a file chooser and returns the name of the file that is selected from the chooser (see “filedialog” on page 210).

alrtdialog

alrtdialog *string*

GUI only

Displays a dialog box containing *string*. The Debugger blocks further input until you dismiss this dialog box. This command is useful inside a MULTI script for displaying a message. For information about displaying other dialog boxes, see “dialog” on page 209.

dialog

dialog *name* [*arguments*]

GUI only

Opens the predefined dialog box named *name* and takes arguments *arguments* if the dialog box permits them. To display a list of the currently defined dialog boxes, use the **I D** (lowercase L, uppercase D) command (see “I” on page 115).

At present, the only supported dialog box is named `textinput`. The `textinput` dialog box displays a prompt and a text field for the user to enter a string. Invoking this dialog box requires two arguments: the first is used as the prompt in the dialog box and the second specifies the MULTI command that is run on the string entered by the user.

The `textinput` dialog box is useful if you would like a custom menu item to run a MULTI command that takes a user-specified argument. For example, suppose that when a custom menu item is selected, you would like a dialog box to prompt the user for the name of a function. Suppose further that the `e` command should be run on the function, displaying the location in source where the function is defined (see “e” on page 147). You might specify that the following command is executed when the menu item is selected:

```
> dialog textinput "Go to the definition of function:" e
```

directorydialog

directorydialog [*window_title*]

GUI only

Opens a directory chooser and returns the name of the directory that is selected from the chooser. By default, the title of the window is **Choose Directory**, but you may change it with the *window_title* parameter. This command is useful for interacting with a user while you are running scripts or evaluating preprocessor macros. See also “filedialog” on page 210.

filedialog

filedialog [*button_label* *window_title* [-defaultdir *dir_name*] [-preset *preset_name*] [-filetypes *file_type* [*file_type*]...]]

GUI only

Opens a file chooser and returns the name of the file that is selected from the chooser. This command is useful for interacting with a user while you are running scripts or evaluating preprocessor macros. See also “directorydialog” on page 210.

By default, the dialog box's button is labeled **Select** and the title of the window is **Choose File**. You may change these with the *button_label* and *window_title* parameters.

The `-defaultdir` option specifies that the file chooser displays the *dir_name* directory, where *dir_name* is the directory you specify. If you do not specify the `-defaultdir` option, the file chooser displays the directory of the last file selected from a Debugger file chooser.

The `-preset` option specifies that the file chooser displays all the file types associated with *preset_name* in its file type drop-down list. Some common *preset_names* and their corresponding file types are:

- Debugger — Lists **All Files**, **Executable**, and **Shared Libraries** in the file type drop-down list.

- `Editable` — Lists **C Source**, **C++ Source**, **Assembly Source**, **Link Map**, **Green Hills Script**, **Configuration File**, **Java Source**, and **Text Files** in the file type drop-down list.
- `Editor` — Lists **All Files** and all file types listed for `Editable` in the file type drop-down list.
- `Object` — Lists **All Files**, **Object Files**, **Shared Libraries**, and **Libraries** in the file type drop-down list.

The `-filetypes` option adds individual file types to the preset. If `file_type` contains a space, enclose it in quotation marks. If you specify more than one `file_type`, separate each with a space. Some common `file_types` and their corresponding file types are:

- `All Files` — Lists **All Files (*)** in the file type drop-down list.
- `Assembly Source` — Lists **Assembly Source (*.s, *.asm, *.si, *.86, *.arm, *.thm, *.68, *.cf, *.mip, *.ppc, *.sh, *.800, *.850, *.830, *.810, *.bf)** in the file type drop-down list.
- `C Source` — Lists **C Source (*.c, *.h)** in the file type drop-down list.
- `C++ Source` — Lists **C++ Source (*.cc, *.cxx, *.cpp, *.cp, *.c++, *.C, *.h, *.hh, *.H, *.h++, *.hxx, *.hpp)** in the file type drop-down list.
- `Libraries` — Lists **Archive (*.a, *.lib, *.olb, *.a88)** in the file type drop-down list.

For more information about preset options and file types, refer to Part II, “Configuring the MULTI IDE” in the *MULTI: Managing Projects and Configuring the IDE* book.

External Tool Commands

The commands in this section deal with external components and sockets.

The following list provides a brief description of each external tool command. For a command's arguments and for more information, see the page referenced.

- **evaltosocket** — Sends the output of the specified command(s) to any socket connections on the server(s) started via the **socket -global** command or the **-socket** command line option (see “evaltosocket” on page 212).

- **make** — Executes the system command **make** to build a target (see “make” on page 212).
- **socket** — Starts a socket server, or closes a socket connection (see “socket” on page 212).

evaltosocket

evaltosocket *commands*

Sends the output of *commands*, which may consist of one or more commands, to any socket connections on the server(s) started by the:

- **socket -global** *port_number*|**auto** command, but not to socket connections started with the **-private_output** option (see “socket” on page 212)
- **-socket** command line option to MULTI (for more information, see Appendix C, “Command Line Reference” in *MULTI: Debugging*)

The output is not sent to the command pane.

make

make [*make_target*]

Executes the system command **make** to build the target *make_target*. If you do not specify a target, the name of the executable you are debugging is used. The output of **make** appears in an Editor window.

socket

socket [-private_output] [-global] *port_number* | auto

socket -close_this

The first format of this command starts a socket server, which allows an external process to send commands to the Debugger and receive output from the Debugger. For example, if you started MULTI on a machine named *myhost* and issued the command **socket** 40000, you could run the command **telnet myhost 40000** to connect a **telnet** window to the Debugger. From the **telnet** window, you could

enter commands that would be executed in the Debugger and receive output from the Debugger. You are not limited to using **telnet**; you can run any program that connects to your host via the specified port and interacts with the Debugger.

The second format of this command closes a socket connection to the socket server.

Available options are:

- `-private_output` — Only sends command output back to the socket connection that was the source of command. Otherwise, command output is sent to the command pane and to all socket connections on the socket server.
- `-global` — Sends output from all Debugger components to the socket connections on the socket server. By default, input from a socket connection on the socket server is sent to the first Debugger component. To send input to a different Debugger component, see “route” on page 201 and “components” on page 110.

When `-global` is not specified, the socket server is only associated with the Debugger component that created it, so output from that Debugger component is sent to the socket connections of the socket server.



Tip

The MULTI command line option **-socket** starts a global socket server when the Debugger opens. For more information about this option, see Appendix C, “Command Line Reference” in the *MULTI: Debugging* book.

- `port_number` — Specifies which port number to use.
- `auto` — Allows the system to automatically find a port.
- `-close_this` — Closes the socket connection that the **socket -close_this** command comes from.

History Commands

The commands in this section deal with commands kept in the Debugger history. For information about the command that launches the History window, a debugging and performance analysis tool, see “history” on page 26.

The Debugger maintains a history of the most recent commands entered in the command pane. The number of commands remembered defaults to 256, but can be set with the **history** configuration option (for more information, see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book).

If you open multiple Debugger windows in a single Debugger session, each Debugger window has its own command history, but when you close the Debugger windows, MULTI saves only the history of the last window. When you launch the Debugger, the history list from the previous session is loaded upon startup (i.e., your command history is maintained across debugging sessions). This behavior can be turned off with the **saveCommandHistory** configuration option (for more information, see “Session Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book).



Tip

For a description of keyboard shortcuts that allow you to auto-complete commands based on your command history, display the last command in your command history, etc., see also “Command Pane Shortcuts” in Appendix B, “Keyboard Shortcut Reference” in the *MULTI: Debugging* book.

The following list provides a brief description of each history command. For a command's arguments and for more information, see the page referenced.

- **!** — Re-executes a command (see “!” on page 215).
- **!!** — Re-executes the last command (see “!!” on page 215).
- **backhistory** — Gives the previous command in the command pane history list (see “backhistory” on page 216).
- **forwardhistory** — Gives the next command in the command pane history list (see “forwardhistory” on page 216).
- **h** — Lists or clears the command history (see “h” on page 217).

!

! [*string*] [*args*]

! [*num*] [*args*]

Re-executes the most recent command beginning with *string*, or re-executes the command numbered *num*. Do not put a space between **!** and *string* or between **!** and *num*. If you include a space, the entire string after **!** is treated as *args*. If specified, *args* are appended as arguments to the command.

If neither a string nor a command number is specified, **!** matches the previous command. In any case, MULTI prints out what was substituted. For example:

```
> echo hello
hello
> !echo hello
"!echo" = "echo hello"
hello hello
> ! echo hello
"! " = "echo hello hello"
hello hello echo hello
> foo
Unknown name "foo" in expression.
> ! echo hello
"! " = "foo"
Unknown name "foo" in expression.
```

!!

!! [*args*]

Re-executes the last command.

You can add additional arguments (*args*) to the end of the command. For example, if the most recent command was

```
echo hi
```

and you type

```
> !! bye
```

then the command

```
echo hi bye
```

will be executed.

The space between `!!` and `args` is required. If you do not include a space, `args` are ignored.

backhistory

backhistory

GUI only

Gives the previous command in the command pane history list. This command is intended to be bound to a key (see “Customizing Keys with the keybind Command” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book). By default, the Debugger binds the **UpArrow** key to this command.

forwardhistory

forwardhistory

GUI only

Gives the next command in the command pane history list. This command is intended to be bound to a key (see “Customizing Keys with the keybind Command” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book). By default, the Debugger binds the **DownArrow** key to this command.

h

h [clear | *num*]

This command has three forms:

- **h** — Lists the existing command history. This option corresponds to **Config** → **State** → **Show Command History**.
- **h clear** — Clears the command history.
- **h num** — Lists the most recent *num* entries in the command history.

Hook Commands

The commands in this section allow you to add hooks to Debugger actions, remove hooks, and list hooks.

The following list provides a brief description of each hook command. For a command's arguments and for more information, see the page referenced.

- **addhook** — Adds a hook to a Debugger action (see “addhook” on page 217).
- **clearhooks** — Removes hooks (see “clearhooks” on page 219).
- **listhooks** — Prints a list of current hooks to the Debugger's **Cmd** pane (see “listhooks” on page 220).

addhook

addhook [-order *number*] [-board | -core *number*] -before *action* {*commands*} | -after *action* {*commands*}

Adds a hook to a Debugger action. You must be connected to a target before using this command.

Available arguments are:

- **-order *number*** — Specifies the new hook's order in the sequence of hooks that run for *action*. (When *action* occurs, the hooks for *action* run in order from the hook with the smallest order number to the hook with the largest order

number.) The hook's order is indicated by *number*, which must be a positive integer. You may use the same order numbers for different actions.

If you do not specify `-order number`, an order number is automatically generated for the new hook. The automatically generated order number is larger than the order number of any existing hook for the given action, allowing you to add hooks without assigning order numbers. Hooks added in this fashion are run in the order that you added them.

- `-board` — [default] Does not associate your new hook with a particular CPU. For example, when you download a program to your target, `-board` hooks for downloading run regardless of the CPU you download to.
- `-core number` — Associates your new hook with a particular CPU. The CPU is indicated by *number*, which must be a non-negative integer. The number you specify should be the same as the number your target connection uses to identify the CPU. For example, if you are using the Green Hills Probe, you should use the core ID number printed out by the `tl` command.

This option is useful for initializing registers in multi-core systems. When you download a program to your target, hooks for cores other than the one to which you are downloading are not run, but `-board` hooks are run.

If you do not specify `-board` or `-core number`, `-board` is passed by default.

- `-before action {commands}` — Runs the command list *commands* before *action* happens to your target. Available actions are:
 - `reset`
 - `download` — If your target has an ordinary, non-“early” MULTI board setup script or a legacy debug server setup script, `-before download` hooks run before the setup script. For information about early MULTI board setup scripts, see “Early MULTI Board Setup Scripts with Debugger Hooks” in Chapter 6, “Configuring Your Target Hardware” in the *MULTI: Debugging* book.
- `-after action {commands}` — Runs the command list *commands* after *action* happens to your target. Available actions are:
 - `connect` — Hooks specified with `-after connect` are only meaningful if you add them from an early MULTI board setup script. For information about early MULTI board setup scripts, see “Early MULTI Board Setup

Scripts with Debugger Hooks” in Chapter 6, “Configuring Your Target Hardware” in the *MULTI: Debugging* book.

- `download` — Hooks specified with `-after download` should leave the target in a halted state rather than a running state. If you want to run the downloaded program immediately after downloading it, use the `r` command (see “`r`” on page 171).
- `reset`
- `rominit` — Hooks specified with `-after rominit` are only valid for programs that are run out of ROM or copied from ROM to RAM. When running from ROM, the `rominit` command is executed after the ROM is initialized, just before the first user instruction of `main()`. When copying from ROM to RAM, the `rominit` command is executed just before the first instruction after the ROM-to-RAM copy.

clearhooks

clearhooks [`-order number`] [`-board` | `-core number`] [`-before action` | `-after action`]

Removes hooks. If no arguments are specified, this command removes all Debugger hooks from your target. You must be connected to a target before using this command.

Available arguments are:

- `-order number` — Removes the hook with the order number specified.
- `-board` — Removes `-board` hooks.
- `-core number` — Removes hooks for the CPU indicated by *number*.
- `-before action` — Removes hooks that run before *action*. Available actions are:
 - `reset`
 - `download`
- `-after action` — Removes hooks that run after *action*. Available actions are:
 - `connect`

- download
- reset
- rominit

For example, to remove all hooks that run after reset, you can issue the following command:

```
> clearhooks -after reset
```

To remove only those hooks that are explicitly bound to run on core 2 after download, you can enter the following command:

```
> clearhooks -core 2 -after download
```

If you are not sure what hooks have already been added, it is a good idea to run the **clearhooks** or **listhooks** command before adding a new set of hooks. See “listhooks” on page 220.

listhooks

listhooks [-order *number*] [-board | -core *number*] [-before *action* | -after *action*]

Prints a list of current hooks to the Debugger's **Cmd** pane. The hooks are printed in a manner suitable for passing them as arguments to the **addhook** command (see “addhook” on page 217).

If no arguments are specified, this command prints all Debugger hooks on your target. You must be connected to a target before using this command.

Available arguments are:

- -order *number* — Prints the hook with the order number specified.
- -board — Prints -board hooks.
- -core *number* — Prints hooks for the CPU indicated by *number*.
- -before *action* — Prints hooks that run before *action*. Available actions are:
 - reset

- download
- `-after action`— Prints hooks that run after *action*. Available actions are:
 - connect
 - download
 - reset
 - rominit

The following example uses the **listhooks** command for a Power Architecture target:

```
> clearhooks
> addhook -core 4 -after reset { target rw cpsr 0xd3 }
> addhook -core 4 -after reset { target rw control 0x1 }
> listhooks -after reset
-order 10    -after reset    -core 4    { target rw cpsr 0xd3 }
-order 20    -after reset    -core 4    { target rw control 0x1 }
```

If you are not sure what hooks have already been added, it is a good idea to run the **listhooks** or **clearhooks** command before adding a new set of hooks. See “clearhooks” on page 219.

MULTI-Python Script Commands

The commands in the following section are related to MULTI-Python scripting. See also the *MULTI: Scripting* book.

The following list provides a brief description of each MULTI-Python script command. For a command's arguments and for more information, see the page referenced.

- **python, py**— Executes a Python statement or script (see “python, py” on page 222).
- **pywin** — Opens or closes the **Py Window** (see “pywin” on page 223).

python, py

```
python [ -b | -nb ] -s "Python_statements" | -f Python_script_name [args]
```

```
py [ -b | -nb ] -s "Python_statements" | -f Python_script_name [args]
```

GUI only

Executes a Python statement or script, where:

- **-b** — Indicates that the Python statement or script executes in blocking mode. In this mode, no Debugger commands are executed until the statement or script has completed.
- **-nb** — [default] Indicates that the Python statement or script executes in non-blocking mode. In this mode, subsequent Debugger commands can execute before the statement or script completes.
- **-s "Python_statements"** — Specifies one or more Python statements for execution. Python statements must be enclosed in quotation marks.
- **-f Python_script_name [args]** — Specifies the Python script for execution. If *Python_script_name* contains spaces, enclose it in quotation marks. The *args* option specifies arguments to the Python script.

The **-f Python_script_name** argument does not require you to enter the full Python script path. If you do not enter the full path, the filename is searched for using the default search path (see “Default Search Path for Files Specified in Commands” on page 14). The first command line argument, `sys.argv[0]`, in the executed Python script is the full path to the Python script file. Other arguments (if any) are those you specified with *args*.

**Note**

The **python** and **py** commands function identically.

For information about using Python scripts to run MULTI, see Chapter 2, “Introduction to the MULTI-Python Integration” in the *MULTI: Scripting* book.

pywin

pywin [-close]

GUI only

Opens or closes the **Py Window**. For information about the **Py Window**, see “MULTI-Python Interfaces” in Chapter 2, “Introduction to the MULTI-Python Integration” in the *MULTI: Scripting* book.

If no option is specified, the **Py Window** appears. If `-close` is specified, the **Py Window** closes.

Object Structure Awareness (OSA) Commands

The commands in this section are intended to be used with Object Structure Awareness packages. For more information, see Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book.

The following list provides a brief description of each OSA command. For a command's arguments and for more information, see the page referenced.

- **osacleanup** — Cleans up OSA information or removes OSA support in order to lessen or eliminate the effect of the OSA package on system performance (see “osacleanup” on page 224).
- **osacmd** — Sends a quoted list of commands to the corresponding OSA package (see “osacmd” on page 225).
- **osaexplorer** — Opens an **OSA Explorer** on the current process in a freeze-mode debugging environment or on the current debug server in a run-mode debugging environment (see “osaexplorer” on page 225).
- **_osaFillGuiWithObj** — Fills in a widget with OSA object attribute values (see “_osaFillGuiWithObj” on page 226).
- **osainject** — Injects the specified message to the specified object (see “osainject” on page 227).
- **osasetup** — Tells MULTI where to find a customized OSA package (see “osasetup” on page 227).

- **osatask** — Opens the Debugger on the task specified (see “osatask” on page 228).
- **osatasklist** — Lists and optionally attaches to OSA tasks (see “osatasklist” on page 229).
- **osaview** — Opens the OSA Object Viewer (see “osaview” on page 229).
- **taskwindow** — Opens the Task Manager in a run-mode debugging environment or displays it in the foreground if it is already open (see “taskwindow” on page 230).

osacleanup

osacleanup [-disable]

GUI only

Cleans up OSA information or removes OSA support. This lessens or eliminates the effect of the OSA package on system performance, thereby speeding up operations like single-stepping.

With no arguments, this command deletes OSA entries from the target list, closes the **OSA Explorer**, and cleans up basic OSA information. The command operates in the context of the selected target list entry. For example, if a freeze-mode connection is selected in the target list, issuing **osacleanup** deletes freeze-mode OSA tasks and closes the **OSA Explorer** that was launched in freeze mode. OSA information in run mode is unaffected.

If you issue the command with no arguments, you are able to restart the OSA package at any time.

If you specify `-disable`, this command removes OSA support for the program. To restart the OSA package for the program, you must restart the program or disconnect from the target and then reestablish the freeze-mode connection. `-disable` is not supported with run-mode connections.

Windows that use the OSA package, such as the **ThreadX Information** window and the multi-core **Task Manager**, do not automatically close when you issue this command.

This command is not supported in TimeMachine or INTEGRITY TimeMachine.

osacmd

osacmd "*OSA_package_commands*"

GUI only

Sends the quoted list of commands to the corresponding OSA package. The Debugger treats the command list as a string. In other words, the command list is not parsed by the Debugger but is sent as is to the OSA package.

osaexplorer

osaexplorer [-refresh]

osaexplorer [-tabname *object_name*] [-tabidx *tab_index*] [-refname *reference_name*]
[-refidx *reference_index*] [-mslrow *row_index*]

GUI only

Opens an **OSA Explorer** on the current process in a freeze-mode debugging environment or on the current debug server in a run-mode debugging environment. The **OSA Explorer** shows information for objects recognized by the OSA integration module. Each attribute of an object is shown as a column in the **OSA Explorer**. For more information about the **OSA Explorer**, see “The OSA Explorer” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book.

Optional arguments to this command are:

- **-refresh** — Refreshes the object list in the **OSA Explorer**. This option is valid only in freeze-mode debugging. The **-refresh** option is not valid if specified in conjunction with any other optional argument.
- **-tabname *object_name*** — Displays the tab *object_name* as the current tab in the **OSA Explorer**. Each tab is named for a particular kind of object. Available **OSA Explorer** tabs appear in the GUI.
- **-tabidx *tab_index*** — Displays the tab specified by *tab_index* as the current tab. Tab indexing starts from zero (0). The left-most tab in the **OSA Explorer** window has a *tab_index* of 0, the next tab a *tab_index* of 1, and so on.

- `-refname reference_name`— Displays the reference list *reference_name* for the object selected in the master pane. The reference list is shown in the reference pane. Available references appear in the drop-down list located above the **OSA Explorer** reference pane.
- `-refidx reference_index`— Displays the reference list specified by *reference_index* in the reference pane. Reference indexing starts from zero (0). The first reference object located in the reference pane's drop-down list has a *reference_index* of 0, the next reference object a *reference_index* of 1, and so on. Any separator present in the drop-down list is included in index numbering.
- `-mslrow row_index`— Selects the master pane row specified by *row_index* and updates the reference pane to show the reference object list for the selected row. Row indexing starts from zero (0). The first row located in the master pane has a *row_index* of 0, the next row a *row_index* of 1, and so on.

You can specify tab, reference, and row arguments together if they make sense in the corresponding **OSA Explorer**. For example, if you are using an **OSA Explorer** with the INTEGRITY operating system, you can enter:

```
> osaexplorer -tabname Task -mslrow 2 -refname "Other Activities"
```

to open an **OSA Explorer** that displays the **Task** tab, selects the third row in the master pane (indexing starts from 0), and shows the task's **Other Activities** in the reference pane.

See also “taskwindow” on page 230.

Corresponds to: **View** → **OSA Explorer**

`_osaFillGuiWithObj`

`_osaFillGuiWithObj` -Widget *widget_name* -ObjType *object_type_name* -ObjFld *fld1* [*fld2*]...

GUI only

Fills in a widget with OSA object attribute values. The widget must be a `TextField`, `PullDown`, or `MScrollList`.

The command is applicable only in a MULTI dialog script.

osainject

osainject -ObjType *object_type_name* -ObjID *object_id* [*message_string*]

GUI only

Injects the message specified by *message_string* to the specified object (with a certain type and ID). MULTI transfers the message injection request to the corresponding OSA module, which then injects the message into the underlying RTOS. The format of *message_string* is OSA module-dependent.

osasetup

osasetup *osa_name* [-cfg *config_filename*] [-lib *module_name*] [-log *log_file*]

GUI only

Tells MULTI where to find a customized OSA package, where:

- *osa_name* — Specifies the name of the OSA package.
- -cfg *config_filename* — Specifies the name of the package's configuration file. If no configuration file is specified, MULTI uses ***osa_name.osa*** as the configuration filename.

You may optionally include the full path to *config_filename*. If a full path is not specified, MULTI searches for the configuration file in your personal configuration directory first:

- Windows 8/Windows 7/Vista — ***user_dir\AppData\Roaming\GHS***
- Windows XP — ***user_dir\Application Data\GHS***
- Linux/Solaris — ***user_dir/.ghs***

and then looks for it in the MULTI IDE installation directory. For more information, see “Freeze-Mode and OSA Configuration File” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book.

- -lib *module_name* — Specifies the package's shared library. If no library is specified, MULTI uses ***osa_name.dll*** (Windows) or ***osa_name.so*** (Linux/Solaris) as the library name.

You may optionally include the full path to *module_name*. If a full path is not specified, MULTI searches for the shared library in the MULTI IDE installation directory first, and if it does not find it, continues to search in a way defined by the host machine.

- `-log log_file` — Specifies that the communication between MULTI and the OSA package be logged to *log_file*.

osatask

osatask [-exec *executable*] [[-addressSpace *name*] -taskid *task_id*]

osatask [-exec *executable*] [[-addressSpace *name*] -taskname *task_name*]

osatask [-exec *executable*] [*task_id_expr_or_task_name_in_quotes*]

GUI only

Opens the Debugger on the specified task, where:

- `-exec executable` — Specifies the name of the executable that contains the task. If no executable is specified, MULTI tries to find the name of the executable.
- `-addressSpace name` — Specifies the name of the address space that contains the task. If *name* contains a space or other special characters, it should be enclosed in double quotation marks.
- `-taskid task_id` — Specifies the numeric ID of the task as displayed in the freeze-mode **OSA Explorer**.
- `-taskname task_name` — Specifies the name of the task as displayed in the freeze-mode **OSA Explorer**. If *task_name* contains a space or other special characters, it should be enclosed in double quotation marks.
- *task_id_expr_or_task_name_in_quotes* — Specifies the ID of the task as an expression, or specifies the name of the task as displayed in the freeze-mode **OSA Explorer**. The task name must be enclosed in double quotation marks.

When run without any arguments, **osatask** opens the Debugger on the task that is currently executing.

This command is only applicable when you are debugging an RTOS in freeze mode. (See also Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book).

osatasklist

osatasklist [-all] [-attach]

GUI only

Prints a list of OSA tasks to the Debugger command pane and, if `-attach` is specified, attaches to them as well. To operate on all OSA tasks, including hidden tasks, specify `-all`.

This command is only supported in a freeze-mode environment.

osaview

osaview [-context]

GUI only

Opens the OSA Object Viewer. If you specify the `-context` option, the OSA Object Viewer opens on information for the INTEGRITY object (AddressSpace or Task, for example) currently displayed in the Debugger window. If you do not specify an option to this command, it displays information for the entire INTEGRITY target.



Note

This command is only supported if you are debugging a run-mode connection and using INTEGRITY version 10 or later.

For information about the OSA Object Viewer, see “The OSA Object Viewer” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

taskwindow

taskwindow [-refresh]

GUI only

Opens the Task Manager in a run-mode debugging environment or displays it in the foreground if it is already open. This command works with run-mode debug connections such as those used with INTEGRITY and VxWorks.

The Task Manager displays the tasks that are running on the (embedded, multitasking) target. It contains columns of information about each of the tasks. For more specific information, see “The Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

To automatically attach to and begin a debugging session on a task, double-click the task. This is equivalent to the command **attach process_id** (see “attach” on page 18). For more information about task management in run-mode, see Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

Specify the `-refresh` option to refresh the existing Task Manager window.

This command can be used in freeze mode to launch the **OSA Explorer**, but this usage is deprecated. Instead, use the **osaexplorer** command for that purpose (see “osaexplorer” on page 225). For more information about object-aware and task-aware debugging in freeze mode, see Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book.

Corresponds to: **View → Task Manager**

Record and Playback Commands

The commands in this section deal with recording and playing back Debugger commands. The Debugger supports the recording and playing back of command sequences to and from files. The files created are ASCII files and can be edited later.

Only Debugger commands can be recorded. If a GUI action executes a Debugger command, that command is recorded. GUI actions include button presses in the Debugger window and mouse clicks in the source pane.

Make sure to follow these guidelines when using record and playback commands and record files:

- If you use the `> file` or `>> file` command when a recording file is already set, the old recording file will be closed and all subsequent commands will be recorded to the new *file*. See “>” on page 232 and “>>” on page 232.
- Scripts may include other scripts, to a maximum script depth of 500.
- The playback file should not contain any lines that begin with `>` or `<`. (Add a space at the beginning of a line, if necessary).
- Standard language-style comments are supported in command playback files, as in all Debugger input (see “Including Comments in Debugger Commands” on page 13).
- You cannot play back from a file that is open for recording, or record to a file that you are playing back.
- Some commands can cause errors that may abort playback. You can use the **Continue running script files on error** GUI option (or the **continuePlaybackFileOnError** configuration option) to prevent these commands from stopping a playback. For more information about this option, see “Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

You can use MULTI's **-p**, **-r**, **-R**, and **-RO** command line options to record commands and/or output or to read from recorded files on startup. For a description of these options, see Appendix C, “Command Line Reference” in the *MULTI: Debugging* book.

The following list provides a brief description of each record and playback command. For a command's arguments and for more information, see the page referenced.

- `>` — Controls or displays the status of command recording (see “>” on page 232).
- `>>` — Controls or displays the state of screen recording (recording commands and their output) (see “>>” on page 232).
- `<` — Starts command playback from the specified file (see “<” on page 233).

>

> [*file* | t | f | c]

(This command is a right angle bracket.)

Controls or displays the status of command recording, where:

- *file* — Sets the command recording file to *file* and turns on command recording. This option corresponds to: **Config → State → Record Commands**.
- t — Turns on command recording (to the current command recording file).
- f — Turns off command recording (but does not close or reset the command recording file).
- c — Turns off command recording and closes the command recording file. (A new recording file will need to be set before recording can be performed again.) This option corresponds to **Config → State → Stop Recording Commands**.

If no argument is specified, the > command displays the current command recording status.

>>

>> [*file* | t | f | c]

(This command is two right angle brackets.)

Controls or displays the state of screen recording (recording commands and their output), where:

- *file* — Sets the screen output recording file to *file* and turns on screen output recording. This option corresponds to **Config → State → Record Commands + Output**.
- t — Turns on screen output recording (to the current screen output recording file).
- f — Turns off screen output recording (but does not close or reset the screen output recording file).
- c — Turns off screen output recording and closes the screen output recording file. (A new recording file will need to be set before recording can be performed

again.) This option corresponds to **Config → State → Stop Recording Commands + Output**.

If no argument is specified, the >> command displays the current screen output recording status.

<

<*file*

(This command is a left angle bracket.)

Starts command playback from the specified *file*.

The specified file will be searched for using the default search path unless the full path has been specified (see “Default Search Path for Files Specified in Commands” on page 14).

Corresponds to: **Config → State → Playback Commands**

Chapter 16

Search Command Reference

Contents

Search Commands	236
-----------------------	-----

The commands in this chapter allow you to search forward and backward in the source pane or in a file, modify a search, or perform an incremental search.

Searches wrap around the beginning and end of files and obey the current case sensitivity setting.

For information about navigating MULTI windows, see Chapter 11, “Navigation Command Reference” on page 145.

Search Commands

The following list provides a brief description of each search command. For a command's arguments and for more information, see the page referenced. (Note that complete descriptions for some of these commands are located in other chapters.)

- **/** — Searches forward through the current file for the specified string (see “/” on page 237).
- **?** — Searches backward for the specified string (see “?” on page 237).
- **bsearch** — Searches backward in the source pane for the previous occurrence of the specified string and highlights it (see “bsearch” on page 238).
- **chgcse** — Sets the case sensitivity of text searches (see “chgcse” on page 238).
- **completeselection** — Selects the smallest complete expression from the text highlighted in the source pane (see “completeselection” on page 239).
- **dialogsearch** — Opens a search dialog box that allows you to search for text or regular expressions in the Debugger's source pane (see “dialogsearch” on page 239).
- **fsearch** — Searches forward in the source pane (after selected text) for the next occurrence of the specified string, and highlights the string (see “fsearch” on page 239).
- **grep** — Searches for the specified text in open files and, if debugging information is available, in all of the files that make up a program (see “grep” on page 240).
- **isearch** — Starts an incremental search in the window specified (see “isearch” on page 241).
- **isearchadd** — Adds the specified text to the search string and continues an incremental search in the specified window (see “isearchadd” on page 242).

- **isearchreturn** — Causes the Debugger to return to the location that was viewed prior to the last **isearch** command (see “isearchreturn” on page 242).
- **printsearch** — Prints the search string or indicates that there is no search string (see “printsearch” on page 243).
- **showdef** — Searches for a C preprocessor definition for each specified name (see “showdef” on page 299 in Chapter 21, “View Command Reference” on page 291).

/

/ [*string*]

In GUI mode, works in the same way as the **fsearch** command (see “fsearch” on page 239).

In non-GUI mode, searches forward through the current file, from the line after the current line, for *string*. Do not put a space between / and *string*.

For example, the command:

/extern

causes the cursor to jump forward to the string `extern`. You can then find more occurrences of this word by repeatedly issuing /, and then pressing **Enter**.

?

? [*string*]

In GUI mode, works in the same way as the **bsearch** command (see “bsearch” on page 238).

In non-GUI mode, searches backward, from the line before the current line, for *string*. Do not put a space between ? and *string*.

For example, the command:

?extern

causes the cursor to jump backward to the string `extern`. You can then find more occurrences of this string by repeatedly issuing `?`, and then pressing **Enter**.

bsearch

bsearch *string*

GUI only

Searches backward in the source pane for the previous occurrence of *string* and highlights it. If the search reaches the beginning of the file, MULTI beeps and then resumes searching from the end.

If *string* is omitted, the *string* argument from the previous **fsearch**, **bsearch**, or incremental search is used. See also “fsearch” on page 239 and “Incremental Searching” in Chapter 9, “Navigating Windows and Viewing Information” in the *MULTI: Debugging* book.

This command is only available in GUI mode. To search backward in non-GUI mode, use the `?` command (see “?” on page 237).

chgcse

chgcse [0 | 1]

Sets the case sensitivity of text searches, where:

- `chgcse 0` makes all future text searches case-sensitive.
- `chgcse 1` makes all future text searches case-insensitive.
- `chgcse` (without an argument) toggles the current case sensitivity setting.



Note

In case-insensitive mode, typing uppercase characters in a search string temporarily changes the search mode to case-sensitive.

completeselection

completeselection

GUI only

Selects the smallest complete expression from the text highlighted in the source pane. If there is no text selected (highlighted) in the source pane, this command does nothing.

If part of a variable name is selected, **completeselection** selects the entire name. It also selects an entire expression in parentheses. For example, if the selection includes an unmatched left parenthesis, the selection will extend to include the matching right parenthesis if it is on the same line as the end of the selection.

dialogsearch

dialogsearch

GUI only

Opens a search dialog box that allows you to search for text or regular expressions in the Debugger's source pane. This dialog contains options for searching forward and backward and for ignoring case.

For more detailed information about the search dialog box, see “The Source Pane Search Dialog Box” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book.

Corresponds to: **Tools** → **Search**

fsearch

fsearch *string*

GUI only

Searches forward in the source pane (after selected text) for the next occurrence of *string*, and highlights it. If *string* is not found before the end of the file, the Debugger beeps and then resumes searching from the beginning of the file.

If *string* is omitted, the *string* argument used in the previous **fsearch**, **bsearch**, or incremental search is used. See also “bsearch” on page 238 and “Incremental Searching” in Chapter 9, “Navigating Windows and Viewing Information” in the *MULTI: Debugging* book.

grep

grep `[[ -i ] [ -w ] [ -F | -E ] text]`

GUI only

Searches for *text* in open files and, if debugging information is available, in all of the files that make up a program.

If no options are specified, this command opens the **Search in Files** dialog (see “Viewing Search in Files Results” in Chapter 4, “Editing Files with the MULTI Editor” in the *MULTI: Managing Projects and Configuring the IDE* book). Alternatively, you can specify a *text* argument and search options from the command line to achieve most of the same functionality as the **Search in Files** dialog, where:

- *text* — Specifies the string to search for. *text* is treated as a basic regular expression unless the `-F` or `-E` option is used.
- `-i` — Causes the **grep** command to perform a case-insensitive search. (Without this option, **grep** performs a case-sensitive search.)
- `-w` — Causes the **grep** command to perform a whole word search. This means that the matching string must be preceded by a non-word character and followed by a non-word character, where word characters are letters, digits, and the underscore. For example, if you specify this option, a search for `ice` does not match `slice` or `ice__`, but it does match `ice-9`. (Without this option, **grep** finds any matching text.)
- `-F` — Causes *text* to be treated as a fixed string.
- `-E` — Causes *text* to be treated as an extended regular expression. Extended regular expressions allow you to use the special regular expression syntax characters `|`, `+`, and `?`, which do not normally have any special meaning to the **grep** command.

Some search strings may be difficult to specify on the command line because the Debugger may interpret escaped characters differently than expected (for example, `\"word`, an escaped double quotation mark followed by the string `word`). If you encounter such a problem, use the **Search in Files** dialog to specify your search string and options.

The output from the **grep** command is displayed in the **Search in Files Results** window (see “Viewing Search in Files Results” in Chapter 4, “Editing Files with the MULTI Editor” in the *MULTI: Managing Projects and Configuring the IDE* book).

This command works by running the BSD **grep** utility. A copy of BSD **grep** is installed along with the MULTI IDE. However, BSD **grep** is not part of MULTI and is not distributed under the same license as MULTI. For more information about the license under which BSD **grep** is distributed, refer to the file **bsdgrep.txt**, which is located in the **copyright** subdirectory of the IDE installation directory. For information about the search expression format that BSD **grep** uses, refer to the OpenBSD `re_format(7)` man page.

Corresponds to: **Tools** → **Search in Files**

isearch

isearch [+ | -] wid=*num*

GUI only

Starts an incremental search in the window specified by *num*, the window ID number. If an incremental search is already active in that window, the current search string is searched again. A plus sign (+) argument specifies a forward search, and a minus sign (-) causes a backward search. If neither a plus or minus sign are specified, a forward search is performed by default.

This command should not be used from the command pane. Instead, use the **keybind** or **mouse** command to bind this command to a key or mouse press. For more information about the **keybind** and **mouse** commands and window ID numbers, see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

isearchadd

isearchadd wid=*num* *text*

GUI only

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

This command should not be used from the command pane. Instead, use the **keybind** or **mouse** command to bind this command to a key or mouse press. For more information about the **keybind** and **mouse** commands and window ID numbers, see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

isearchreturn

isearchreturn wid=*num*

GUI only

Causes the Debugger to return to the location (in the window specified by the window ID number *num*) that was being viewed prior to the last **isearch** command.

This command is only meaningful after an **isearch** command has been issued (that is, it is only meaningful if the window with the identification number *num* is performing an incremental search).

This command should not be used from the command pane. Instead, use the **keybind** or **mouse** command to bind this command to a key or mouse press. For information about the **keybind** and **mouse** commands and window ID numbers, see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

printsearch

printsearch

Prints the search string or indicates that there is no search string.

If a search string exists, it is printed within square brackets, so the beginning and ending whitespace can be seen. For example:

```
> printsearch
```

might print:

```
[ foo  ]
```

meaning that the search string is the word `foo` preceded by one space and followed by two spaces.

Chapter 17

Target Connection Command Reference

Contents

General Target Connection Commands	246
Serial Connection Commands	260

General Target Connection Commands

The commands in this section allow you to connect to and manipulate a debug target platform.

The following list provides a brief description of each general target connection command. For a command's arguments and for more information, see the page referenced.

- **change_binding** — Associates the currently selected executable with a compatible connection, or disassociates the currently selected executable from a connection (see “change_binding” on page 247).
- **connect** — Connects to a target or modifies the logging of transactions between MULTI and the target (see “connect” on page 247).
- **connectionview** — Opens the **Connection Organizer** window, which allows you to create, edit, and manage Connection Methods (see “connectionview” on page 250).
- **disconnect** — Closes an existing connection to a target (see “disconnect” on page 250).
- **iobuffer** — Disables or enables buffering for the current connection's I/O panes (see “iobuffer” on page 251).
- **load** — Downloads the current executable to the target's memory (see “load” on page 251).
- **prepare_target** — Prepares the target or opens the **Prepare Target** dialog box (see “prepare_target” on page 253).
- **reset** — Resets the target (see “reset” on page 256).
- **set_runmode_partner** — Sets or disables a run-mode partner for the current freeze-mode connection or opens the **Set Run-Mode Partner** dialog box (see “set_runmode_partner” on page 256).
- **setup** — Executes a target setup script (see “setup” on page 257).
- **target, xmit** — Transmits commands directly to the target debug server, and supplies the debug server with the current task context (see “target, xmit” on page 258).
- **targetinput, xmitio** — Feeds a string into target standard input (see “targetinput, xmitio” on page 259).

- **unload** — Unloads a program from the target system's memory (see “unload” on page 259).

change_binding

change_binding bind | unbind

Associates the currently selected executable with a compatible connection, or disassociates the currently selected executable from a connection. If you pass the `bind` option and the executable is only compatible with one connection, it is automatically associated with that connection. If it is compatible with more than one connection, the **Use Which Connection?** dialog box appears. If it is not compatible with any currently available connection, the **Connection Chooser** prompts you to connect to a target.

For more information, see “Associating Your Executable with a Connection” in Chapter 7, “Preparing Your Target” in the *MULTI: Debugging* book.

change_binding unbind corresponds to: **Debug** → **Use Connection** → **Stop Using Current Connection**

connect

connect *connection_method_name*

connect [setup=*filename* [setupargs=*script_arguments*]] [log[=*filename*]] [-temp] *debug_server* [*dbserver_arguments*]

connect -restart_runmode

connect log[=*filename*] | nolog

connect

The first three formats of this command connect or reconnect to a target (a simulator, emulator, monitor, or OS, for example). You must connect to a target before you can perform certain MULTI Debugger operations.

The fourth format of the command starts or stops the logging of transactions between MULTI and the debug server.

The fifth format of the command opens the **Connection Chooser**, which allows you to create, edit, or choose a Connection Method to connect to your target. See “Creating a Standard Connection Using the Connection Chooser” in Chapter 3, “Connecting to Your Target” in *MULTI: Debugging*.

The argument for the first format of this command is:

- `connection_method_name` — Specifies the Connection Method that is used to connect to your target. Passing this argument is equivalent to selecting a Connection Method from the **Connection Chooser's** drop-down list.

Arguments for the second format of this command, which is equivalent to creating a Custom Connection Method in the **Connection Chooser**, are:

- `setup=filename` — Specifies the target setup script. The commands in the specified file will be run before downloading is performed. This argument is optional because not all targets require setup scripts.

The `setup=` option can be used to specify **.mbs**, **.py**, and **.gpy** setup scripts. For more information about setup scripts, see Chapter 6, “Configuring Your Target Hardware” in the *MULTI: Debugging* book.

- `setupargs=script_arguments` — Specifies one or more script arguments to the target setup script `filename` (above). If `script_arguments` contains spaces, enclose the argument string in double quotation marks ("*string with spaces*").

At present, only **.py** and **.gpy** Python setup scripts can accept arguments.

- `log[=filename]` — Specifies that transactions between MULTI and the debug server should be logged and sent to standard error or, if specified, to the file `filename`.
- `-temp` — Prevents the connection from being saved across sessions.
- `debug_server` — Specifies the debug server to use to connect to the target. A debug server is a program that controls the target device and must be designed for the hardware debugging interface you are using (if any) and the target CPU for which you are compiling your program.

- *dbserver_arguments* — Specify debug-server-specific options. For supported options for:
 - INTEGRITY run-mode target connections, see Chapter 4, “INDRT2 (rtserv2) Connections” in the *MULTI: Debugging* book or Chapter 5, “INDRT (rtserv) Connections” in the *MULTI: Debugging* book.
 - Green Hills Probe or SuperTrace Probe target connections, see the *Green Hills Debug Probes User's Guide*.
 - Other target connections, see the *MULTI: Configuring Connections* book for your processor family.

If specified, the `setup`, `setupargs`, `log`, and `-temp` options must appear before `debug_server`.



Note

If the string of arguments to the **connect** command follows the first or second format of this command, MULTI first attempts to exactly match the string to the name of a Connection Method. If MULTI does not find an exact match, it interprets the string as the name of a target debug server and as setup and debug server options (if specified).



Note

The Debugger ignores the deprecated `mode` argument in connections that specify it. Even for connections that do not explicitly include this argument, the Debugger may print a message stating that the `mode` argument is deprecated. This occurs if the `mode` argument has been associated with a MULTI 4 Connection Method whose name matches the arguments of the **connect** command you entered. To remove the `mode` argument from the Connection Method, edit and save the Connection Method in MULTI 7. For more information, see “Updating MULTI 4 Target Connections” in Chapter 7, “Preparing Your Target” in the *MULTI: Debugging* book.

Arguments for the third and fourth formats of this command are:

- `-restart_runmode` — Attempts to reconnect to your last run-mode partner connection, which can be useful for reestablishing lost run-mode partner connections. This option is only meaningful if you have established a run-mode partner during the current debugging session, and you are eligible to connect

to it (that is, you are using a freeze-mode connection, your program is running on the target, and the target supports a run-mode partner). For information about run-mode partners, see “Automatically Establishing Run-Mode Connections” in Chapter 4, “INDRT2 (rtserv2) Connections” in the *MULTI: Debugging* book.

See also “set_runmode_partner” on page 256.

- `log [=filename]` — Starts transaction logging between MULTI and the debug server, and sends the log to standard error or, if specified, to the file *filename*.
- `nolog` — Stops transaction logging and closes the log file. This command is only meaningful if you have previously started transaction logging. See the preceding description of `log=filename`.

Corresponds to: 

Corresponds to: **Target** → **Connect**

connectionview

connectionview [*connection_file*]

GUI only

Opens the **Connection Organizer** window, which allows you to create, edit, and manage Connection Methods. If no filename is specified, the window opens with the **[User Methods]** connection file displayed. Otherwise, the window opens with the specified file displayed. If the specified file does not exist, it is created.

For more information about the **Connection Organizer**, see “Using the Connection Organizer” in Chapter 3, “Connecting to Your Target” in the *MULTI: Debugging* book.

Corresponds to: **Target** → **Show Connection Organizer**

disconnect

disconnect

Closes an existing connection to a target.

Corresponds to: 

Corresponds to: **Target** → **Disconnect from Target**

iobuffer

iobuffer { on | off }

GUI only

Disables or enables buffering for the current connection's I/O pane. Buffering is enabled by default. If buffering is enabled (on), input to the I/O pane is not sent to the target until a newline character is encountered in the input stream. If buffering is disabled (off), every character is sent to the target as soon as it is typed. Disabling the buffering in MULTI may cause problems on some targets if they expect input to be buffered.

Corresponds to: **Target** → **IO Buffering**

load

load [-wait [-global] [-time *milliseconds*]] [-setup | -nosetup] [-nohook board | core | all] [*filename*]

Downloads the current executable to the target's memory. This may take a long time, depending on the size of the program. After being loaded, the program is not started automatically. Whether the `.bss` section is cleared depends on the debug server.

Available arguments are:

- `-wait` — Blocks command processing until the download is complete. Optional arguments to `-wait` are:
 - `-global` — Blocks command processing for all MULTI processes in the same MULTI session. If you do not specify `-global`, only the current process is blocked.
 - `-time milliseconds` — Blocks command processing only up to *milliseconds*. In other words, *milliseconds* represents the maximum

period of time that command processing is blocked, whether or not the download has completed.

- `-setup` — Causes the Debugger to run the setup script specified in the **connect** command before loading the program (see “connect” on page 247). This is the default.



Tip

The **setup** command allows you to execute the setup script without loading a program (see “setup” on page 257).

- `-nosetup` — The Debugger loads the program without running the setup script.
- `-nohook` — Specifies that the Debugger should not run download hook scripts, where:
 - `board` — Skips running download hooks that are not associated with any CPU (that is, hooks created with the **addhook -board** command).
 - `core` — Skips running download hooks that are associated with a CPU (that is, hooks created with the **addhook -core** command).
 - `all` — Skips running all download hooks.

For information about the **addhook** command, see “addhook” on page 217.

- `filename` — If you are connected to a hardware target, the named file will be downloaded to the target's memory without changing the image that is currently open in the Debugger. Use this option with extreme caution. MULTI will assume that the named file contains an adequate subset of the image that is open in the Debugger, and will attempt to execute and debug it as such, without attempting to download the current image as well. Ordinarily, if you want to change which executable you are debugging, you should issue a **debug** command to change the image that is open in the Debugger (see “debug” on page 21), then click the **Prepare Target** button () to download it.

The filename you specify will be searched for using the default search path. See “Default Search Path for Files Specified in Commands” on page 14.

On a run-mode connection to an INTEGRITY target, the following additional restrictions apply:

- You should specify the filename of an INTEGRITY application.
- The setup script will not be executed, and you should not pass the `-setup` or `-nosetup` options.
- Loading applications on INTEGRITY is only supported if the target supports and was configured with a dynamic loader (for example, the LoaderTask).
- Whether the program is started automatically depends on the `StartIt` settings in the Integrate configuration file for the application.

Some targets support interruptible downloads. To interrupt a download in progress, press **Esc**.

See also “prepare_target” on page 253.

prepare_target

prepare_target [`-ask` | `-flash` | `-load` [`-wait` [`-time` *milliseconds*] [`-global`]] | `-verify=sparse` | `-verify=complete` | `-verify=none`] [`-allcores` | `-onecore`] [`-save` | `-nosave`]

Prepares the target by downloading, flashing, or verifying one or more executables, or opens the **Prepare Target** dialog box so that you can specify a download, flash, or verify operation. Available options are:

- `-ask` — Opens the **Prepare Target** dialog box.
- `-flash` — Loads the program onto the target as if the **flash gui** command were used (see “flash” on page 134). Issuing `prepare_target -flash` corresponds to setting **Program Flash ROM** in the **Prepare Target** dialog box.
- `-load` — Loads the program onto the target as if the **load** command were used (see “load” on page 251). Issuing `prepare_target -load` corresponds to setting **Download to RAM** in the **Prepare Target** dialog box.
- `-wait` — If you are using **-load** to dynamically download a module over a run-mode connection, blocks command processing until the download is complete. Otherwise, this option has no effect.

Optional arguments to `-wait` are:

- `-time milliseconds` — Blocks command processing only up to *milliseconds*. In other words, *milliseconds* represents the maximum period of time that command processing is blocked, whether or not the download has completed.
- `-global` — Blocks command processing for all MULTI processes in the same MULTI session. If you do not specify `-global`, only the current process is blocked.
- `-verify=sparse` — Checks the target's memory against the contents of the executable file as if the **verify -sparse -all** command were used (see “verify” on page 143). Issuing `prepare_target -verify=sparse` corresponds to setting **Program already present on target. Verify: Sparsely** in the **Prepare Target** dialog box.
- `-verify=complete` — Checks the target's memory against the contents of the executable file as if the **verify -all** command were used (see “verify” on page 143). Issuing `prepare_target -verify=complete` corresponds to setting **Program already present on target. Verify: Completely** in the **Prepare Target** dialog box.
- `-verify=none` — Does *not* check the target's memory against the contents of the executable file. You should use this option only when you are sure that it is safe to proceed assuming that the contents of the executable file match the contents of your target's memory. Issuing `prepare_target -verify=none` corresponds to setting **Program already present on target. Verify: Not At All** in the **Prepare Target** dialog box.



Note

When you use any of the `prepare_target -verify` options or **Program already present on target. Verify** settings, and **Automatically use these settings for this program next time** is checked in the **Prepare Target** dialog box, subsequent attempts to reload or restart the program will only verify that your target's memory matches the contents of the executable. If you need to reload the program onto your target, use the `-load` or `-flash` options, or change the dialog box setting to **Download to RAM** or **Program Flash ROM**.

- `-allcores` — Specifies that when you download, flash, or verify the currently selected executable on the core it is associated with, each remaining core of your multi-core target is automatically prepared as if you had run `prepare_target -verify=none` on the executable associated with it. Note that the `-allcores` option is deprecated. We recommend that you instead use one of the methods described in “Providing Configuration Details to the Debugger” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in *MULTI: Debugging*.
- `-onecore` — Specifies that when you download, flash, or verify the currently selected executable on the core it is associated with, executables associated with the remaining cores of your multi-core target are ignored. Note that this option is deprecated. We recommend that you instead use one of the methods described in “Providing Configuration Details to the Debugger” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in *MULTI: Debugging*.
- `-save` — Automatically uses these settings the next time you:
 - Pass this command without options or
 - Click the **Prepare Target** button ()for the currently selected executable.
- `-nosave` — Does not automatically use these settings (see `-save` above). Instead, MULTI opens the **Prepare Target** dialog box. This is the default behavior.

If you do not specify any options, MULTI either performs the operation(s) last executed when the executable was selected, performs a default operation (based on the type of program you are debugging), or opens the **Prepare Target** dialog box to receive input.

Regardless of the option you specify, MULTI opens the **Prepare Target** dialog box if input is required.

If you have not connected to a target prior to issuing this command, the **Connection Chooser** opens.

This command may not be available for use with relocatable modules.

For more information, see “Preparing Your Target” in Chapter 7, “Preparing Your Target” in the *MULTI: Debugging* book.

Corresponds to: 

Corresponds to: **Debug** → **Prepare Target**

reset

reset [halt | run | hold]

Resets the target, where:

- **halt** — Causes the Debugger to wait for the target to halt and leaves the target in the halted state after reset. This is the default behavior if you do not specify any option to the **reset** command. To stop waiting and to abort the command, press **Esc**.
- **run** — Runs the target after performing the reset. This option also disables all hardware breakpoints.
- **hold** — Causes the target to keep asserting the reset signal.

This command is only available for hardware targets, and not every target supports all types of reset. Some targets may need to emulate the **halt** or **run** behaviors (for example, by performing the reset and then performing either the halt or run action immediately thereafter).

Corresponds to: 

set_runmode_partner

set_runmode_partner [-none | -auto | -reset | *Connection_Method_name*]

Sets or disables a run-mode partner for the current freeze-mode connection or opens the **Set Run-Mode Partner** dialog box so you can change the setting via the GUI. For information about run-mode partners, see “Automatically Establishing Run-Mode Connections” in Chapter 4, “INDRT2 (rtserv2) Connections” in the *MULTI: Debugging* book.

Available options are:

- **-none** — Disables the run-mode partner functionality for the current freeze-mode connection.

- `-auto` — Specifies that the operating system should attempt to tell the Debugger what address and method to use to establish a run-mode connection to the target. This option is only supported with certain operating systems (such as INTEGRITY IoT and INTEGRITY version 10 and later).
- `-reset` — Resets the run-mode partner setting to the default, unset state.
- `Connection_Method_name` — Sets `Connection_Method_name` as the run-mode partner. If `Connection_Method_name` does not exist when the Debugger tries to initialize the run-mode partner, an error is printed.

If you do not specify any options, the **Set Run-Mode Partner** dialog box is displayed. See “The Set Run-Mode Partner Dialog Box” in Chapter 4, “INDRT2 (rtserv2) Connections” in the *MULTI: Debugging* book.

See also the `-restart_runmode` option in “connect” on page 247.

setup

setup [-first] [-args *script_arguments*] [*script_filename*]

Executes a target setup script, where:

- `-first` — Treats the setup script as if it has never been executed on the target. This option clears the `_ALREADY_SETUP_ONCE` system variable. Whether or not this affects the execution of your setup script depends on the setup script itself.
- `-args script_arguments` — Specifies script arguments to the target setup script. At present, only Python setup scripts can accept arguments.
- `script_filename` — Specifies the target setup script to be executed. If you do not specify `script_filename`, the target setup script associated with the debug connection is executed.

To interrupt this command, press **Esc**.



Note

If the setup script relies on program symbols, and the symbol context is separate from the hardware context, setup script execution will fail. To successfully execute the setup script, use the **prepare_target** command to run the setup script and load the program directly, or use

prepare_target -verify=none to merge the hardware and symbol contexts, and then use the **setup** command to run the setup script. For more information, see “prepare_target” on page 253.



Note

The **setup** command is not supported with legacy (**.dbs**) scripts. Use the command **target script script_filename** instead (see “target, xmit” on page 258).

See also “connect” on page 247 and “load” on page 251.

target, xmit

target [/NoRmtMsg] *string*

xmit [/NoRmtMsg] *string*

Transmits commands directly to the target debug server, and supplies the debug server with the current task context. You can change the current task context with the **route** command (see “route” on page 201).

Using the **target** or **xmit** command is equivalent to entering *string* in the Debugger target pane, where *string* is one or more supported debug server commands.

In GUI mode, the first message from the target (while it executes the *string* commands) will be printed in the current Debugger window's command pane by default. The /NoRmtMsg option directs MULTI to print the message in the target pane instead.

For a list of commands that can be passed to your debug server (if any), see the *MULTI: Configuring Connections* book for your target processor family.



Note

The Debugger cannot predict the effect of a **target** or **xmit** command. If the command changes the state of the target, you may have to take corrective action to cause the new state of the target to be reflected in the Debugger. For example, you may need to issue the **halt** or **update** command after the **target** or **xmit** command completes. See “halt” on page 170 and “update” on page 301.

targetinput, xmitio

targetinput [input_string_to_target]

xmitio [input_string_to_target]

Feeds a string into the target standard input. The input string can be a plain string or be enclosed in double quotation marks. Special characters can be sent in an escape sequence that begins with a backslash (\), as in C. For example, a new line can be sent with the sequence: \n.

The **targetinput** and **xmitio** commands are subject to the same limitations as the Debugger's I/O pane. For more information, see “The I/O Pane” in Chapter 2, “The Main Debugger Window” in the *MULTI: Debugging* book.

unload

unload [-wait [-time *milliseconds*] [-global]] [-filedialog | *filename*]

Unloads a program from the target system's memory.

Available arguments are:

- **-wait** — Blocks command processing until the specified program has been unloaded. Optional arguments to **-wait** are:
 - **-time *milliseconds*** — Blocks command processing only up to *milliseconds*. In other words, *milliseconds* represents the maximum period of time that command processing is blocked, whether or not the program has been unloaded.
 - **-global** — Blocks command processing for all MULTI processes in the same MULTI session. If you do not specify **-global**, only the current process is blocked.
- **-filedialog** — Opens a file chooser from which you can select a file to unload from the target (assuming the file is already loaded). This is the default.
- ***filename*** — Unloads the specified file (for example, *a.out*) if it matches a program on the target.

This command is only supported on INTEGRITY and VxWorks run-mode connections.

Serial Connection Commands

The commands in this section allow you to control MULTI's serial terminal emulator (**MTerminal**). For more information, see Chapter 30, “Establishing Serial Connections” in the *MULTI: Debugging* book.

The following list provides a brief description of each serial connection command. For a command's arguments and for more information, see the page referenced.

- **serialconnect** — Establishes a connection to a serial port (see “serialconnect” on page 260).
- **serialdisconnect** — Terminates a previously established serial connection (see “serialdisconnect” on page 261).

serialconnect

serialconnect *port_name* [-baud *baudrate*] [-databits *DB*] [-parity *P*] [-stopbits *SB*] [-flowcontrol *FC*]

Establishes a connection to a serial port, where:

- *port_name* — Specifies which serial port is being used (for instance, `tttya`, `ttys0`, or `COM1`).
- -baud *baudrate* — Specifies the baud rate, where *baudrate* can be any one of the following: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, or 230400. The default is 9600.
- -databits *DB* — Specifies the data bits, where *DB* can be 5, 6, 7, or 8. The default is 8.
- -parity *P* — Specifies parity, where *P* can be `none`, `even`, or `odd`. The default is `none`.
- -stopbits *SB* — Specifies stop bits, where *SB* can be either 1 or 2. The default is 1.
- -flowcontrol *FC* — Specifies flow control, where *FC* can be `none` or `xonxoff`. The default is `none`.

Corresponds to: **Tools** → **Serial Terminal** → **Make Serial Connection**

serialdisconnect

serialdisconnect

Terminates a previously established serial connection.

Corresponds to: **Tools** → **Serial Terminal** → **Disconnect from Serial**

Chapter 18

Task Group Command Reference

Contents

Task Group Commands	264
---------------------------	-----

The commands in this chapter allow you to operate on task groups. Task groups allow you to organize tasks, making it easier to work with multiple tasks simultaneously. For information about task groups, see “Working with Task Groups in the Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

**Note**

For these commands to work properly, the Task Manager must be open. For information about opening the Task Manager, see “The Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.

Task Group Commands

The following list provides a brief description of each task group command. For a command's arguments and for more information, see the page referenced.

- **changegroup** — Adds tasks to a task group or removes tasks from a task group (see “changegroup” on page 264).
- **creategroup** — Creates a task group (see “creategroup” on page 266).
- **destroygroup** — Destroys the specified task groups (see “destroygroup” on page 267).
- **groupaction** — Runs, halts, or single-steps all tasks that belong to the specified task groups (see “groupaction” on page 267).
- **listgroup** — Lists task groups (see “listgroup” on page 268).

changegroup

```
changegroup -add | -del @task_group [-addressSpace AddressSpace_name]  
[-taskname] task_name[, task_name]... | [-taskid] task_id[, task_id]...
```

GUI only

Adds tasks to a task group or removes tasks from a task group, where:

- **-add** — Adds the specified tasks into the specified task group.
- **-del** — Deletes the specified tasks from the specified task group.

- `@task_group` — Specifies the task group that tasks should be added to or deleted from. If `task_group` contains spaces, enclose it in quotation marks.
- `-addressSpace AddressSpace_name` — Specifies the INTEGRITY AddressSpace where the tasks are located. You only need to give the AddressSpace if you specify task names that are not unique. On INTEGRITY, every task ID is unique.
- `[-taskname] task_name[, task_name]...` — Specifies the tasks to operate on by task name.
- `[-taskid] task_id[, task_id]...` — Specifies the tasks to operate on by task ID.

If you do not specify `-taskname` or `-taskid`, MULTI treats numeric values as task IDs and other values as task names.

You can add/delete multiple tasks from the same AddressSpace or add/delete tasks from multiple AddressSpaces. If you want to operate on tasks from multiple AddressSpaces and the tasks share the same name, format the command as shown below:

```
> changegroup -add @"My Group" -addressSpace AddressSpace1 \
Initial -addressSpace AddressSpace2 Initial
```

For information about task groups, see “Working with Task Groups in the Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.



Note

The Task Manager must be open when you execute this command.

creategroup

creategroup *@task_group* [-addressSpace *AddressSpace_name*] [-taskname] *task_name*[, *task_name*]... | [-taskid] *task_id*[, *task_id*]...

GUI only

Creates a task group, where:

- *@task_group* — Specifies the task group name. If *task_group* contains spaces, enclose it in quotation marks.
- *-addressSpace AddressSpace_name* — Specifies the INTEGRITY AddressSpace where the tasks are located. You only need to give the AddressSpace if you specify task names that are not unique. On INTEGRITY, every task ID is unique.
- *[-taskname] task_name[, task_name]...* — Specifies the names of tasks you want to add to the task group.
- *[-taskid] task_id[, task_id]...* — Specifies the IDs of tasks you want to add to the task group.

If you do not specify *-taskname* or *-taskid*, MULTI treats numeric values as task IDs and other values as task names.

You can add multiple tasks from the same AddressSpace or add tasks from multiple AddressSpaces. If you want to add tasks from multiple AddressSpaces and the tasks share the same name, format the command as shown below:

```
> creategroup @"My Group" -addressSpace AddressSpace1 Initial \  
-addressSpace AddressSpace2 Initial
```

For information about task groups, see “Working with Task Groups in the Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.



Note

The Task Manager must be open when you execute this command.

destroygroup

destroygroup *@task_group1* [, *@task_group2*]...

GUI only

Destroys the specified task groups. If there are spaces in a task group name, enclose it in quotation marks.

For information about task groups, see “Working with Task Groups in the Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.



Note

The Task Manager must be open when you execute this command.

groupaction

groupaction -r|-h|-s *@task_group* [, *@task_group*]...

GUI only

Runs (-r), halts (-h), or single-steps (-s) all tasks that belong to the specified task groups. If *task_group* contains spaces, enclose it in quotation marks.

As long as the target operating system supports task groups, these actions will be performed on the individual tasks synchronously. If an operating system does not support task groups, MULTI will send out separate commands to each task in the task group. In this case, the latency time for the operations on different tasks will be unpredictable, depending on various factors such as network traffic, the RTOS debug agent's status, and the target's speed. For more information, see “Legacy Method of Group Execution and Control” in Chapter 22, “Run-Mode Debugging” in *MULTI: Debugging*.

For information about task groups, see “Working with Task Groups in the Task Manager” in Chapter 22, “Run-Mode Debugging” in *MULTI: Debugging*. For information about using this command in a freeze-mode environment, see “Legacy Methods of Group Execution and Control” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in *MULTI: Debugging*.



Note

The Task Manager must be open when you execute this command.

listgroup

listgroup [-d] [@task_group1 [, @task_group2]...]

GUI only

Lists task groups. If you do not specify any arguments, all existing task groups are listed. If you specify `-d`, MULTI lists detailed information about the task groups. Specify one or more task groups to see information only about those task groups. If there are spaces in a task group name, enclose it in quotation marks.

For information about task groups, see “Working with Task Groups in the Task Manager” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book.



Note

The Task Manager must be open when you execute this command.

Chapter 19

TimeMachine and INTEGRITY TimeMachine Command Reference

Contents

INTEGRITY TimeMachine Commands	270
TimeMachine and Trace Commands	271

The commands in this chapter allow you to use TimeMachine and INTEGRITY TimeMachine. For more information, see Part IV, “TimeMachine and INTEGRITY TimeMachine Debugging” in the *MULTI: Debugging* book.

INTEGRITY TimeMachine Commands

The following list provides a brief description of each INTEGRITY TimeMachine command. For a command's arguments and for more information, see the page referenced. (Note that the complete description of one of these commands is located in another section.)

- **timemachine** — Enables or disables TimeMachine or INTEGRITY TimeMachine, or launches Separate Session TimeMachine (see “timemachine” on page 272).
- **tmhistory** — Undoes or redoes execution in INTEGRITY TimeMachine (see “tmhistory” on page 270).

tmhistory

tmhistory - | +

Undoes or redoes execution in INTEGRITY TimeMachine, where:

- - — Returns to the previous location in the history of places where INTEGRITY TimeMachine has halted. This can be useful if you want to undo an action (such as running backwards in the INTEGRITY TimeMachine Debugger) that brought you to an unexpected location in your source code. The **tmhistory -** command corresponds to **TimeMachine → Go to Previous Time**.
- + — Returns to the next location in the history of places where INTEGRITY TimeMachine has halted. This option is only meaningful if you have previously issued the **tmhistory -** command. The **tmhistory +** command corresponds to **TimeMachine → Go to Next Time**.

This command causes INTEGRITY TimeMachine to run through the log data to the new destination. INTEGRITY TimeMachine proceeds to the selected destination ignoring all breakpoints, but otherwise behaves as it does when it is running as a result of any other run or step command.

TimeMachine and Trace Commands

The commands in this section allow you to use TimeMachine or collect, analyze, and save trace data for use with the TimeMachine tool suite.

The following list provides a brief description of each TimeMachine and trace command. For a command's arguments and for more information, see the page referenced.

- **timemachine** — Enables or disables TimeMachine or INTEGRITY TimeMachine, or launches Separate Session TimeMachine (see “timemachine” on page 272).
- **trace** — Starts a new trace session or modifies an existing trace session (see “trace” on page 273).
- **tracebrowse** — Launches a trace browser for the specified address expression (see “tracebrowse” on page 277).
- **tracedata** — Configures trace so that the trace trigger occurs when the specified data address is read from or written to (see “tracedata” on page 277).
- **tracefunction** — Configures trace so that trace data is only collected when the process is executing the specified function (see “tracefunction” on page 278).
- **traceline** — Configures trace so that the trace trigger occurs when the specified address is executed (see “traceline” on page 278).
- **traceload** — Loads the previously saved trace session file you specify (see “traceload” on page 278).
- **tracemevsys** — Generates an EventAnalyzer log from the current trace data and opens the **EventAnalyzer** on the information (see “tracemevsys” on page 279).
- **tracepath** — Generates path analysis information from the current trace data and opens a **PathAnalyzer** window on the information (see “tracepath” on page 280).
- **tracepro** — Generates profiling information from the current trace data and opens a **Profile** window on the information (see “tracepro” on page 280).
- **tracesave** — Saves the trace session to the specified file (see “tracesave” on page 280).

- **tracesavetext** — Saves the currently retrieved trace data to the specified text file (see “tracesavetext” on page 281).
- **tracesubfunction** — Configures trace so that trace data is only collected when the process is executing an address within the specified function or when the process is executing a callee of the specified function (see “tracesubfunction” on page 282).

timemachine

timemachine [-newsession | -ns] [-tid *task_ID* | -as_name *AddressSpace*]

GUI only

Enables or disables TimeMachine for the specified target list entry, enables or disables INTEGRITY TimeMachine for all target list entries, or launches Separate Session TimeMachine on the specified target list entry, where:

- no arguments — If you have licensed the TimeMachine Debugger and you used the SuperTrace Probe or a Green Hills simulator to collect trace over a freeze-mode connection, this enables/disables TimeMachine for the selected target list entry.

If you have licensed the INTEGRITY TimeMachine Debugger and you instrumented your virtual AddressSpaces for INTEGRITY TimeMachine debugging over a run-mode connection, this enables/disables INTEGRITY TimeMachine for all target list entries.

- **-newsession** and **-ns** — Launch Separate Session TimeMachine on the selected target list entry or on the entry specified by **-tid** or **-as_name**. The **-newsession** and **-ns** options function identically. For information about Separate Session TimeMachine, see “Using Separate Session TimeMachine” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in *MULTI: Debugging*.
- **-tid** — Enables/disables TimeMachine for the task with the specified *task_ID*.
- **-as_name** — Enables/disables TimeMachine for the specified *AddressSpace*.

For more information about using TimeMachine, including prerequisites, see Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in *MULTI*:

Debugging. For more information about using INTEGRITY TimeMachine, including prerequisites, see Chapter 19, “Debugging with INTEGRITY TimeMachine” in *MULTI: Debugging*.

Corresponds to: 

Corresponds to: **TimeMachine** → **TimeMachine Debugger/TimeMachine** → **Leave TimeMachine Mode**

trace

trace [abort] [bookmarks] [clear] [close] [config=*filename*] [disable | enable] [history - | history +] [list] [options] [path] [pro | profiler] [reg | register] [retrieve [-all]] [set [*option* [*value*]]] [stats] [sync[on | off]] [toggle] [triggers] [updateosa] [api *application_name* [*application_arguments*...]]

Starts a new trace session or modifies an existing trace session.

The available arguments are:

- **abort** — Aborts the retrieval of trace data. This option corresponds to **TimeMachine** → **Abort Trace Retrieval**.
- **bookmarks** — Opens the **Trace Bookmarks** window. For more information, see “Bookmarking Trace Data” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.
- **clear** — Clears all current trace data on the host, trace probe, and target. This option corresponds to **TimeMachine** → **Clear Data**.
- **close** — Clears trace data and closes all windows associated with trace. This argument overrides other specified options.
- **config=*filename*** — Specifies the name of a saved trace configuration file to load.
- **disable** — Stops trace collection and retrieves trace data. This option corresponds to **TimeMachine** → **Disable Trace**.
- **enable** — Starts trace collection and clears any previously collected data on the target. Data that has already been retrieved is not cleared, but if trace retrieval is currently in progress, it is aborted. This option corresponds to **TimeMachine** → **Enable Trace**.

- `history -` — Returns to the previous location in the trace navigation history. This can be useful if you want to undo an action (such as running backwards in the TimeMachine Debugger) that brought you to an unexpected location in your source code.

The **trace history -** command corresponds to  (see “Pre-Defined Buttons” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book).

- `history +` — Returns to the next location in the trace navigation history. This option is only meaningful if you have previously issued the **trace history -** command (described above).

The **trace history +** command corresponds to  (see “Pre-Defined Buttons” in Appendix A, “Debugger GUI Reference” in the *MULTI: Debugging* book).

- `list` — Opens the Trace List. For more information, see “Viewing Trace Data in the Trace List” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book. This option corresponds to **TimeMachine → Trace List**.
- `options` — Opens the **Trace Options** window, which allows you to modify options related to trace data collection and display. For more information, see “The Trace Options Window” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book. This option corresponds to **TimeMachine → Trace Options**.
- `path` — Opens the **PathAnalyzer**. For more information, see “The PathAnalyzer” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book. This option corresponds to **TimeMachine → PathAnalyzer**.
- `pro` or `profiler` — Generates profiling data from the current trace data and opens a **Profile** window on the information. The **trace pro[filer]** command functions identically to the **tracepro** command (see “tracepro” on page 280). For more information, see “Using Trace Data to Profile Your Target” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book. This option corresponds to **TimeMachine → Profile**.
- `reg` or `register` — Opens the **Reconstructed Registers** window. The `reg` and `register` options function identically. For more information, see “Viewing Reconstructed Register Values” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.
- `retrieve` — Retrieves trace data from the trace probe or target.

With SuperTrace Probe v3 targets, this either retrieves the amount of data set by the **Target buffer size** option, or it retrieves twice as much data as has already been retrieved. In the latter case, all previously retrieved trace data is cleared from the tools and then retrieved again from the probe. For more information, see “Retrieving Trace Data from a SuperTrace Probe v3” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

With all other targets, all trace data is always retrieved.

This option corresponds to **TimeMachine** → **Retrieve Trace**.

- `retrieve -all` — Retrieves all collected data from the SuperTrace Probe v3. Prior to retrieving the data, all trace data in the tools is cleared. For more information, see “Retrieving Trace Data from a SuperTrace Probe v3” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.
- `set [option]` — Lists all available target-specific trace options and their current values, or lists only the target-specific option *option* and its current value. For information about options, see the documentation about target-specific trace options in the *Green Hills Debug Probes User's Guide* or, if you are using a V850 target, the documentation about V850 trace options in the *MULTI: Configuring Connections* book.
- `set option value` — Sets the target-specific trace option *option* to *value*. For information about options, see the documentation about target-specific trace options in the *Green Hills Debug Probes User's Guide* or, if you are using a V850 target, the documentation about V850 trace options in the *MULTI: Configuring Connections* book. You can also set target-specific trace options in the **Trace Options** window. For information about this window, see “The Trace Options Window” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book.
- `stats` — Opens the **Trace Statistics** window. For more information, see “Viewing Trace Statistics” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book. This option corresponds to **TimeMachine** → **Trace Statistics**.
- `sync` — Prints the status of packet selection synchronization. See the description of `sync on`.

- `sync on` — Enables cross-core synchronization of trace packets based on time. This is the default.

This option is only supported on targets that support multi-core trace. If you attempt to enable trace synchronization on a target that does not support it, the message `Unable to set synchronizer state` is printed. This option has no effect unless the trace data is timestamped. For more information, see “Using Trace Tools on a Multi-Core Target” in Chapter 25, “Freeze-Mode Debugging and OS-Awareness” in the *MULTI: Debugging* book.

- `sync off` — Disables cross-core synchronization of trace packets.
- `toggle` — Toggles collection of trace data. This argument overrides other specified options.
- `triggers` — Opens the **Set Triggers** window, which allows you to specify triggers and other trace events. For more information, see “The Set Triggers Window” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book. This option corresponds to **TimeMachine** → **Set Triggers**.
- `updateosa` — Refreshes the OSA data used for task-aware trace of an operating system. This argument is intended to be used when the **Assume static OSA** trace option is enabled. It is useful for forcing an update of OSA data when the set of tasks in the system has reached a static state. It can also be useful for refreshing data after the state of the system has been changed, for example, by downloading an application using a run-mode debug server such as `rtserv` or `rtserv2`. For more information, see the description of the **Assume static OSA** option in “The Collection Tab” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book.
- `api application_name [application_arguments...]` — Launches a C/C++ application or a Python script that uses the live TimeMachine interface. You may optionally supply one or more `application_arguments`. All arguments passed after `application_name` are treated as application arguments, not as arguments to the **trace** command. For more information, see “The TimeMachine API” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.



Note

The `toggle` and `close` arguments override other specified options.

tracebrowse

tracebrowse [-line] [address_expression]

GUI only

Launches a trace browser for the specified address expression. If the address expression refers to a data address, a **Trace Memory Browser** will be launched to display reads and writes to the address. If the address expression refers to a function, a **Trace Call Browser** will be launched to display the call sites of the function. If -line is specified, *address_expression* refers to a line number in the current function. In this case, a **Trace Instruction Browser** will be launched to display the executions of that particular line. If no address is specified, the current function is displayed in the **Trace Call Browser**.

For more information about the trace browsers, see “Browsing Trace Data” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

Trace data must be collected before this command can be used.

tracedata

tracedata [address_expression]

Configures trace so that the trace trigger occurs when the data address specified by *address_expression* is read from or written to. Using this command overwrites any triggers or trace events that were previously set.

On INTEGRITY, you can only use this command to set the trigger in the kernel AddressSpace.

For information about triggers, see “Configuring Trace Collection” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book.

tracefunction

tracefunction [*address_expression*]

Configures trace so that trace data is only collected when the process is executing the function specified by *address_expression*. To collect trace data for both the specified function and its callees, use the **tracesubfunction** command (see “tracesubfunction” on page 282).

Using this command overwrites any triggers or trace events that were previously set.

On INTEGRITY, you can only use this command to configure trace collection in the kernel AddressSpace.

For more information, see “Configuring Trace Collection” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book.

traceline

traceline [*address_expression*]

Configures trace so that the trace trigger occurs when the address specified by *address_expression* is executed. If no address is specified, the currently selected line is used. Using this command overwrites any triggers or trace events that were previously set.

On INTEGRITY, you can only use this command to set the trigger in the kernel AddressSpace.

For information about triggers, see “Configuring Trace Collection” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book.

traceload

traceload [*filename*]

Loads the previously saved trace session file *filename*. If you do not specify *filename*, you are prompted to make a selection in the file chooser that appears.

If the file was saved without an ELF file and you are using INTEGRITY, you must load the file from the kernel executable. If the file was saved without an ELF file and you are not using INTEGRITY, you can only load the file while debugging the same program you used to gather the trace data. If you have rebuilt the program since you collected the trace data, loading the saved trace data may produce unexpected behavior.

You cannot load a trace session file into an earlier version of MULTI than was used to save the file. This is true of major versions only. For example, if a file was saved in MULTI 7.x, you cannot use MULTI 6.x to load the file; you must use MULTI 7.x or later.

For more information, see “Saving and Loading a Trace Session” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

tracemevsys

tracemevsys [-file *description_file*] [-no_task_name]

GUI only

Generates an EventAnalyzer log from the current trace data and opens the **EventAnalyzer** on the information. The *description_file* argument must specify a trace system call description file, which is specific to the operating system for which the EventAnalyzer log is being generated.

If the `-no_task_name` option is specified, task names are not read from the target. By default, reading task names is enabled, which halts the target momentarily if the target is not already halted. If this option is specified, the task ID is used as the task name when the **EventAnalyzer** is displayed.

See “Viewing Trace Events in the EventAnalyzer” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

Corresponds to: **TimeMachine** → **EventAnalyzer**

tracepath

tracepath

GUI only

Generates path analysis information from the current trace data and opens a **PathAnalyzer** window on the information.

See also “The PathAnalyzer” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

Corresponds to: **TimeMachine** → **PathAnalyzer**

tracepro

tracepro

GUI only

Generates profiling data from the current trace data and opens a **Profile** window on the information. See “Using Trace Data to Profile Your Target” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

Corresponds to: **TimeMachine** → **Profile**

tracesave

tracesave [--data | -d | --data_elf | -de | --data_elf_debug | -ded | --win | -w] [--scratch_dir=*path* | -s=*path*] [--overwrite | -o] [*filename*]

Saves a trace session, where:

- --data | -d — Saves trace data only (not the ELF file or debug information).
- --data_elf | -de — Saves the trace data and ELF file only (not the debug information).
- --data_elf_debug | -ded — [default] Saves the trace data, ELF file, and debug information.

- `--win | -w` — Opens a dialog box allowing you to choose what type of information is saved.
- `--scratch_dir=path | -s=path` — Specifies the path to the directory where temporary files created during the process of saving the trace session are stored. By default, they are stored in the directory where the trace session file is created. If there is not enough space on that file system for both the temporary files and the trace session file, use this option to specify an alternative location for the temporary files.
- `--overwrite | -o` — Overwrites the existing file *filename* without prompting you.
- *filename* — Specifies the file that the trace session is saved to. If you do not specify *filename*, you are prompted to make a selection in the file chooser that appears.

You can load *filename* with the **traceload** command (see “traceload” on page 278).

For more information, see “Saving and Loading a Trace Session” in Chapter 20, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book.

tracesavetext

tracesavetext *filename*

Saves the currently retrieved trace data to the text file *filename*. The data is saved in a Comma-Separated-Value format suitable for analysis by custom scripts.



Tip

The recommended interface for custom trace analysis scripts is the TimeMachine API. It is much more powerful and flexible and enables higher performance custom analysis.

Corresponds to: **File** → **Export As CSV** in the Trace List

tracesubfunction

tracesubfunction [*address_expression*]

Configures trace so that trace data is only collected when the process is executing an address within the function specified by *address_expression* or when the process is executing a callee of the function. To collect trace data for the specified function but not for callees, use the **tracefunction** command (see “tracefunction” on page 278).

Using this command overwrites any triggers or trace events that were previously set.

On INTEGRITY, you can only use this command to configure trace collection in the kernel AddressSpace.

For more information, see “Configuring Trace Collection” in Chapter 21, “Advanced Trace Configuration” in the *MULTI: Debugging* book.

Chapter 20

Tracepoint Command Reference

Contents

Tracepoint Commands	284
---------------------------	-----

The commands in this chapter allow you to manipulate tracepoints. These commands are only available if tracepoints are supported on the currently connected target. For more information about tracepoints, see Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

Tracepoint Commands

The following list provides a brief description of each tracepoint command. For a command's arguments and for more information, see the page referenced.

- **edittp** — Opens the **Tracepoint Editor** dialog box, which allows you to edit the tracepoint at the current source line (see “edittp” on page 284).
- **passive** — Toggles passive mode on or off or changes the passive mode password (see “passive” on page 285).
- **tpdel** — Deletes a tracepoint (see “tpdel” on page 285).
- **tpenable** — Enables or disables a tracepoint (see “tpenable” on page 286).
- **tplist** — Lists the current tracepoints (see “tplist” on page 286).
- **tpprint** — Collects the current data buffer from the target and displays the data in the command pane as ASCII text (see “tpprint” on page 287).
- **tppurge** — Clears the entire tracepoint buffer on the target (see “tppurge” on page 287).
- **tpreset** — Resets the hit count for a tracepoint (see “tpreset” on page 288).
- **tpset** — Sets a tracepoint (see “tpset” on page 288).

edittp

edittp

GUI only

Opens the **Tracepoint Editor** dialog box, which allows you to edit the tracepoint at the current source line. If no tracepoint is set on the current line (as indicated by the blue context arrow), you can use the dialog to create a new tracepoint. For more information, see “Tracepoint Editor Dialog” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

passive

passive [on | off]

passive [on | off] *password*

passive password *old_pw new_pw*

passive toggles passive mode on or off. In passive mode, the Debugger rejects invasive debugging that significantly impacts program functionality. Passive mode is not available with all targets.

- *password* — Enter the passive mode password for operating system integrations that require a password.

passive password changes the passive mode password. This use of the **passive** command only has meaning for operating system integrations that support passive mode passwords. The following must be specified immediately after the **passive password** command.

- *old_pw* — Enter the old passive mode password.
- *new_pw* — Enter the new passive mode password.

For more information about passive mode and the **passive** command, see “Debugging in Passive Mode” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tpdel

tpdel [*address_expression* | %*id*]

Deletes the tracepoint at the specified *address_expression* or with the specified tracepoint identification number *id*. If no argument is given, the tracepoint at the current line is removed. See also “Deleting a Tracepoint” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tpenable

tpenable { true | false } [*address_expression* | %*id*]

Enables or disables the tracepoint at the specified *address_expression* or with the specified tracepoint identification number *id*.

If `false` is specified, the tracepoint will not collect data until the tracepoint is re-enabled. Depending on the target operating system, each time a tracepoint is encountered, even if it is disabled, there may be a small processing overhead. If `true` is specified, the tracepoint will actively collect data.

See also “Enabling or Disabling a Tracepoint” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tplist

tplist [verbose | quiet] [refresh]

Lists the current tracepoints (whether enabled or disabled). The information listed includes the tracepoint identification number, the location and address, the “hit count / timeout” threshold, and if available, the number of times the tracepoint has been reached.

If `quiet` (the default) is specified, this command displays the tracepoints in a one-per-line format. If `verbose` is specified, this command displays the data gathering command in addition to the location and type information.

Normally, this command displays the information as of the last time the Debugger contacted the target; however, if `refresh` is specified, the Debugger will contact the target for current information. Note that refreshing the list requires transmitting information from the target and may impact the target's execution. By default, this command does not perform a `refresh`.

A sample output of the **tplist** command might be:

```
> tplist
  0 main#2:      0x101f4 200/400 (argc,argv)
```

This output indicates that a tracepoint with identification number 0 is set at the address 0x101f4 to collect the values of variables `argc` and `argv`. The tracepoint will be disabled by the target if it is hit more than 200 times per 400 time units.

Using the **verbose** option provides additional information about the exact actions the tracepoint will perform. For example:

```
> tplist verbose
  0 main#2:      0x101f4 200/400
  (argc) :  READ_MEM RELATIVE 0001 0000000c 00 04 00000001
  (argv) :  READ_MEM RELATIVE 0001 00000008 00 04 00000001
```

See also “Listing Tracepoints” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tpprint

tpprint [file=*filename*]

Collects the current data buffer from the target and displays the data in the command pane as ASCII text. If *filename* is specified, the data will be written to that file instead of being displayed in the command pane. Note that *filename* must be quoted. Additionally, if *filename* contains a backslash, you must escape it with another backslash. See also “Viewing the Tracepoint Buffer” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tppurge

tppurge [all]

Clears the entire tracepoint buffer on the target. The optional `all` argument is allowed for compatibility with previous releases, but it has no effect.

See also “Purging the Tracepoint Buffer” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tpreset

tpreset [*address_expression* | %*id*]

Resets the hit count for the tracepoint at the specified *address_expression* or with the specified tracepoint identification number *id*. If no argument is given, the tracepoint at the current line is reset. See also “Resetting a Tracepoint” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

tpset

tpset *count* / *timeout* (*variable_list*) [*address_expression*] [[*condition*]]

Sets a tracepoint, where:

- *count* — Specifies a hit count.
- *timeout* — Specifies the timeout threshold. The units of the *timeout* period are determined by the target's implementation of tracepoints.
- (*variable_list*) — Specifies a comma-separated list of variables whose values are collected by the tracepoint. You must enclose the variable list in parentheses.
- *address_expression* — Specifies the location.
- [*condition*] — Specifies a condition, whose interpretation is based on the target's implementation of tracepoints. You must enclose the condition in square brackets.

To reduce performance impact, the target automatically disables tracepoints that are hit more than *count* times per *timeout* time units. If you do not want tracepoints to be automatically disabled, specify 0 for both *count* and *timeout*.

The following example sets a tracepoint that collects the values of `argc` and `argv` at the second source line of `main()`. If the tracepoint is hit more than 200 times per 400 time units, it will be automatically disabled. (The exact length and definition of the time units used by tracepoints is implementation-specific. For more information, consult the documentation for your operating system integration.)

```
> tpset 200/400 (argc,argv) main#2
  0 main#2:      0x101f4 200/400
```

```
(argc) :  READ_MEM RELATIVE 0001 0000000c 00 04 00000001  
(argv) :  READ_MEM RELATIVE 0001 00000008 00 04 00000001
```

The output from the **tpset** command describes the actual operations that take place when the tracepoint is triggered. In the above example, the target will read register 0001 and add an offset of 0000000c when gathering data for the variable `argc`. The target will then read 00000001 blocks of 04 bytes and store that value into the tracepoint buffer.

See also “Setting a Tracepoint” in Chapter 28, “Non-Intrusive Debugging with Tracepoints” in the *MULTI: Debugging* book.

Chapter 21

View Command Reference

Contents

General View Commands	292
Cache View Commands	305
Data Visualization Commands	306

General View Commands

The following list provides a brief description of each general view command. For a command's arguments and for more information, see the page referenced. (Note that complete descriptions for some of these commands are located in other chapters.)

- **bpview, breakpoints** — Opens the **Breakpoints** window (see “bpview, breakpoints” on page 52 in Chapter 3, “Breakpoint Command Reference” on page 43).
- **browse** — Opens a browser for the specified object type (see “browse” on page 294).
- **browseref, xref** — Displays the specified object's cross references in a Browse window or MULTI's command pane (see “browseref, xref” on page 295).
- **callsviiew** — Opens the **Call Stack** window, which lists all functions on the call stack (see “callsviiew” on page 79 in Chapter 5, “Call Stack Command Reference” on page 77).
- **connectionview** — Opens the **Connection Organizer** window, which allows you to create, edit, and manage Connection Methods (see “connectionview” on page 250 in Chapter 17, “Target Connection Command Reference” on page 245).
- **diff** — Opens the **Diff Viewer** on two versions of the same file or on two different files (see “diff” on page 296).
- **edit** — Opens an Editor on the file and line specified (see “edit” on page 296).
- **editview** — Opens a MULTI Editor for the object specified (see “editview” on page 297).
- **heapview** — Opens the **Memory Allocations** window (see “heapview” on page 297).
- **localsview** — Displays the current function's local variables in a Data Explorer (see “localsview” on page 298).
- **memview** — Opens a new **Memory View** window for displaying and modifying memory contents (see “memview” on page 299).
- **noteview** — Navigates to the location of the specified Debugger Note or opens a Note Browser displaying all Debugger Notes for the program being debugged (see “noteview” on page 104 in Chapter 7, “Debugger Note Command Reference” on page 101).

- **osaview** — Opens the OSA Object Viewer (see “osaview” on page 229 in Chapter 15, “Scripting Command Reference” on page 197).
- **regview** — Opens a **Register View** window displaying all registers, or opens a **Register Information** window displaying the specified register (see “regview” on page 195 in Chapter 6, “Configuration Command Reference” on page 83).
- **showdef** — Searches for a C preprocessor definition for each specified name (see “showdef” on page 299).
- **showhistory** — Displays the specified source file's revision history in the **History Browser** (see “showhistory” on page 300).
- **taskwindow** — Opens the Task Manager in a run-mode debugging environment or displays it in the foreground if it is already open (see “taskwindow” on page 230 in Chapter 15, “Scripting Command Reference” on page 197).
- **top** — Opens a **Process Viewer** window, which displays a snapshot of the processes on your native target (see “top” on page 300).
- **update** — Re-evaluates all currently open Data Explorer and monitor windows, halting the process if necessary to get the updated information (see “update” on page 301).
- **view** — Displays the given items in a Data Explorer, a Browse window, or an OSA Object Viewer (see “view” on page 301).
- **viewdel** — Closes all Data Explorer, Browse, **Register View**, **Memory View**, **Call Stack**, and **Breakpoints** windows (see “viewdel” on page 303).
- **viewlist** — Displays a list of structures in the Data Explorer (see “viewlist” on page 303).
- **window** — Creates, deletes, lists, or changes the contents of a monitor window (see “window” on page 304).

browse

browse [*object_type*]

GUI only

Opens a browser for *object_type*. The following list describes available *object_types*, of which you may specify only one. If no argument is given, *funcs* is assumed. Passing an unknown object causes a Data Explorer to be opened on the object.

- *files* | *filelist* — Opens a Browse window listing all files in the program. See “Browsing Source Files” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- *funcs* | *functions* — Opens a Browse window listing all functions in the program. See “Browsing Functions” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- *global* | *globals* — Opens a Browse window listing all the global variables in the program. See “Browsing Global Variables” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- *types* — Opens a Browse window displaying all structs, classes, and unions used in the program. See “Browsing Data Types” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- *file* — Opens a Browse window listing all the functions in the file *file*.
- *includes* [*file* | *all*] — Opens a Graph View window that displays an include file dependency graph. If you specify *file*, the graph is centered on the given file. If you specify *all*, the entire program's include file dependency graph is shown (note that this may be very large for many programs). If you do not specify either of these options, the graph is centered on the current file being viewed in the source pane. See “Browsing Includes” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- *classes* — Opens a Tree Browser for classes. See “Browsing Classes” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- *class* — Opens a Data Explorer listing data members and functions of the class *class*.

- `caller [function]` — Opens a Browse window displaying the callers of the function *function*, if specified. Otherwise, it displays the callers of the current function being viewed in the source pane.
- `callee [function]` — Opens a Browse window displaying the callees of the function *function*, if specified. Otherwise, it displays the callees of the current function being viewed in the source pane.
- `scalls | calls [function]` — Opens a Tree Browser for static calls, rooted on the function *function*, if specified. Otherwise it is rooted on the current function being viewed in the source pane. See “Browsing Static Calls By Function” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.
- `fcalls [file]` — Opens a Tree Browser for static calls between files, rooted on the file *file*, if specified. Otherwise, it is rooted on the current file being viewed in the source pane. See “Browsing Static Calls By File” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.

browseref, xref

browseref [-all | -write | -read | -addr | -nowindow] *object_name*

xref [-all | -write | -read | -addr | -nowindow] *object_name*

GUI only

Displays the specified object's cross references in a Browse window or, if `-nowindow` is specified, in MULTI's command pane. `-all`, which is the default setting, displays all cross references; `-write` displays all writes; `-read` displays all reads; and `-addr` displays all address references. See also Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book.

The **browseref** and **xref** commands attempt to resolve *object_name* to a symbol in your program as if *object_name* were used in an expression, searching for it based on the position of the current line pointer and using the scope rules of the language in use. For more information, see “Expression Scope” in Chapter 14, “Using Expressions, Variables, and Function Calls” in the *MULTI: Debugging* book.

**Note**

For performance reasons, these commands only partially implement compatible type checking when searching for references in compile units other than the one where the search started. This means that two very similar structures in two different compile units can be incorrectly matched as identical. In C++, MULTI can match the wrong structure if two similar nested structures exist. Nameless types exacerbate this problem, so ensuring that all types are named reduces the chance that this will happen.

diff

diff [*filename*]

GUI only

Opens the **Diff Viewer** on the local and repository versions of *filename* or, if a filename has not been specified, on the local and repository versions of the file currently displayed in the source pane. The file must be under a supported version control system; otherwise, the **Choose two files** dialog box appears, prompting you to select a second file to diff against the first.

edit

edit [*address_expression*]

GUI only

Opens an Editor on the file and line given by *address_expression*. If no *address_expression* is given, this command opens an Editor on the currently displayed location.

For 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 also “Using Address Expressions in Debugger Commands” on page 5.

Corresponds to:

editview

editview [*expr* | *func* | *file*]

GUI only

Opens a MULTI Editor for the object specified by *expr*, *func*, or *file*, where:

- *expr* — Is an expression.
- *func* — Is a function name.
- *file* — Is a filename.

For functions and files, the **editview** command opens a MULTI Editor window that contains the specified source code. For expressions and variables, this command opens a Data Explorer that contains the given expression. You can then use the Data Explorer to edit it. You can bind this command to a mouse button to create a “smart” mouse click that views or edits anything you click. For more information, see “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

heapview

heapview [showleaks [-new] | showallocations [-new] | setmark | showstats | showvisual]

GUI only

Opens the **Memory Allocations** window. This window's operations are only available for processes that are halted and runnable. For more information, see “Using the Memory Allocations Window” in Chapter 16, “Viewing Memory Allocation Information” in the *MULTI: Debugging* book.

The available arguments are:

- **showleaks** — Causes the window to open with the **Leaks** tab displayed. If **-new** is specified with **showleaks**, only leaks created since the last mark command are shown.

- `showallocations` — Causes the window to open with the **Allocations** tab displayed. If `-new` is specified with `showallocations`, only current allocations created since the last `mark` command are shown.
- `setmark` — Marks all objects in the heap.
- `showstats` — Causes the window to open with the **Visualization** tab displayed, and causes the tab to show heap statistics.
- `showvisual` — Causes the window to open with the **Visualization** tab displayed.

If no `show*` option is specified, the window opens with the **Visualization** tab displayed by default.

Corresponds to: **View** → **Memory Allocations**

localsview

localsview

GUI only

Displays the current function's local variables in a Data Explorer. For more information, see “Displaying Local Variables” in Chapter 11, “Viewing and Modifying Variables with the Data Explorer” in *MULTI: Debugging*.

This is equivalent to the **view \$locals\$** command (see “view” on page 301).

Corresponds to: 

Corresponds to: **View** → **Local Variables**

memview

memview `[@count] address_expression`]

GUI only

Opens a new **Memory View** window for displaying and modifying memory contents, where:

- `@count` — Forces the **Memory View** window to display at most `count` bytes. The minimum value that can be specified is 4.

This argument is useful if you do not want to manually size the window, but you do want to restrict the amount of data displayed to a small, exact number of bytes. It is less useful for showing a large set of data. If you set `count` to a higher number of bytes than the **Memory View** window can display, only the number of bytes that fit in the window are shown.

- `address_expression` — Specifies that the contents of the **Memory View** window begin with the memory address `address_expression`. If no address expression is specified, an empty **Memory View** window opens.

As an example use of this command, suppose you want to open a **Memory View** window sized to show 128 bytes, beginning with the address `argv[0]`. You would enter:

```
> memview @128 argv[0]
```

For more information, see Chapter 15, “Using the Memory View Window” in the *MULTI: Debugging* book.

Corresponds to: 

Corresponds to: **View** → **Memory**

showdef

showdef `[name]...`

Searches for a C preprocessor definition for each specified name. Each definition found is printed. **showdef** will not find definitions that come from assembly files.

If no arguments are specified, this command prints all the preprocessor definitions in the current program. In both cases, the local definitions list is searched first, then the global definitions list is searched. Every name that has a C preprocessor definition anywhere in the current program has an entry in the global definitions list. Any name that has more than one C preprocessor definition in the program has an overriding definition in the local definitions list for any files that use the non-global definition. This command is only enabled for programs built with MULTI debugging information.

showhistory

showhistory [*address_expression*]

GUI only

Displays the revision history in the **History Browser** for the source file specified by *address_expression*. If no address expression is specified, this command displays the revision history for the file corresponding to the currently displayed location.

For more information about address expressions, see “Using Address Expressions in Debugger Commands” on page 5.

top

top

GUI only

Opens a **Process Viewer** window, which displays a snapshot of the processes on your native target, much like the Linux/Solaris `top` or `ps` utilities.

See “Viewing Native Processes” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

Corresponds to: **View** → **Task Manager**

update

update *[[-m] interval]*

GUI only

Forces all currently open Data Explorer and monitor windows (including **Register View**, **Memory View**, **Call Stack**, and OSA Object Viewer windows) to be re-evaluated, halting the process if necessary to get the updated information. If this command must perform a halt, the process resumes after the windows are refreshed. This command provides you with a way to update your Data Explorer to the current values without requiring you to manually halt and resume the process.

If you specify an interval, MULTI automatically updates the windows, again halting and resuming the process if necessary. The update occurs approximately every *interval* seconds or, if you pass *-m*, approximately every *interval* milliseconds. This is a useful way to monitor the value of a variable continuously. To deactivate the automatic update, specify 0 for the *interval*.

Corresponds to: **View** → **Refresh Views**

view

view *[/n | /m] [/data] expr [, expr]... | type | *address | filename | \$locals\$ | number:\$locals\$ | number:var*

GUI only

Displays the given items in a Data Explorer, a Browse window, or an OSA Object Viewer.

If specified, the following options must appear before any other options on the command line.

- */n* — Forces the specified items to open in a new Data Explorer. This is not relevant for INTEGRITY objects unless */data* is also used.
- */m* — Forces the specified items to open in the primary Data Explorer (the one whose title bar says **Data Explorer**). This is not relevant for INTEGRITY objects unless */data* is also used.

- `/data` — Forces the specified variable to open in a Data Explorer rather than in an OSA Object Viewer. This is only relevant for INTEGRITY objects.

The `/n` option overrides the `/m` option. The `/data` option may be specified in conjunction with the `/n` option or the `/m` option.

The remaining options are:

- `expr [, expr] ...` — Displays the specified expression(s) `expr` in a Data Explorer.¹ If the expression is for an INTEGRITY object, the expression appears in an OSA Object Viewer instead.
- `type` — Displays the type `type` in the Data Explorer.
- `*address` — Displays the contents of the given memory location in the Data Explorer. You must precede the address with an asterisk (*).
- `filename` — Displays `filename`'s functions in the Browse window.
- `$locals$` — Displays the current function's local variables in the Data Explorer. For more information, see “Displaying Local Variables” in Chapter 11, “Viewing and Modifying Variables with the Data Explorer” in *MULTI: Debugging*.

This option corresponds to the  button, **View → Local Variables**, and the **localsview** command (see “localsview” on page 298).

- `number:$locals$` — Displays local variables for the function located `number` levels up the stack. The information is displayed in the Data Explorer. For a fixed view of the current stack frame, enter:

```
> view 0:$locals$
```

For more information, see “Displaying Local Variables” in Chapter 11, “Viewing and Modifying Variables with the Data Explorer” in *MULTI: Debugging*.

- `number:var` — Displays the local variable `var` for the function located `number` levels up the stack. The information is displayed in the Data Explorer. The stack frame does not change when the program counter (PC) moves to a new function.

¹Viewing expressions with side effects (such as `x++`) is not recommended. Side effects will occur whenever the Debugger evaluates the expression, which it may do several times each time the Data Explorer is updated, and at other times as well. This can lead to unpredictable behavior in the Data Explorer.

For more information about the window in which a given item may be displayed, see one of the following:

- “The Data Explorer Window” in Chapter 11, “Viewing and Modifying Variables with the Data Explorer” in the *MULTI: Debugging* book
- “The Browse Window” in Chapter 12, “Browsing Program Elements” in the *MULTI: Debugging* book
- “The OSA Object Viewer” in Chapter 22, “Run-Mode Debugging” in the *MULTI: Debugging* book

Corresponds to: **View** → **View Expression**

viewdel

viewdel

GUI only

Closes all of the current Data Explorer, Browse, **Register View**, **Memory View**, **Call Stack**, and **Breakpoints** windows.

Corresponds to: **View** → **Close All Views**

viewlist

viewlist *structptr nextptr [links]*

GUI only

Displays a list of structures in the Data Explorer, where *structptr* is the pointer to the structure, *nextptr* is the name of the pointer to the next element of the structure, and *links* is the number of items in the list to be displayed (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 display the first three items in this list. In this case, the **viewlist** command is equivalent to entering:

```
view ptr; view ptr->next; view ptr->next->next;
```

window

window [*num*] [{*commands*}]

GUI only

Creates, deletes, lists, or changes the contents of a monitor window. A monitor window captures the output of a command or command list (see “Using Command Lists in Debugger Commands” on page 12). Each time the process stops, the commands are executed, and the output of these commands is printed in the monitor window. The command list may contain multiple commands separated by a semicolon (;), and multiple commands must be surrounded by curly braces (for example, `window {calls; B}`).

This command has several forms.

- **window** — Lists all existing monitor windows and their assigned commands.
- **window num** — Deletes monitor window number *num*. The number is displayed on the monitor window border. For example, entering `window 2` removes the monitor window named `MONITOR 2`.
- **window {commands}** — Creates a monitor window displaying the results of the given command list.
- **window num {commands}** — Replaces the command list for monitor window *num* with *commands*. To change the command list, left-click the command's name in the upper-left corner of the monitor window.
- **window 0** — (The number zero.) Deletes all existing monitor windows.

For example, assuming you have not used any monitor windows in your current debugging session, the command `window calls` displays a stack trace in monitor window **MONITOR 1**. To change the window to display the breakpoints, use the command `window 1 B`. See also “monitor” on page 29.

Cache View Commands

The commands in this section allow you to view the contents of the caches on your processor. These commands are only available on some processors. For more information about viewing caches, see “Viewing Caches” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

The following list provides a brief description of each cache view command. For a command's arguments and for more information, see the page referenced.

- **cachefind** — Opens the **Cache Find** window (see “cachefind” on page 305).
- **cacheview** — Opens the **Cache View** window (see “cacheview” on page 305).

cachefind

cachefind [address_expression]

GUI only

Opens the **Cache Find** window. If you specify an address expression, the **Cache Find** window displays cache information for the corresponding address. For more information, see “The Cache Find Window” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

Corresponds to: **View** → **Find Address in Cache**

cacheview

cacheview

GUI only

Opens the **Cache View** window. For more information, see “The Cache View Window” in Chapter 18, “Using Other View Windows” in the *MULTI: Debugging* book.

Corresponds to: **View** → **Caches**

Data Visualization Commands

The commands in this section allow you to control custom data visualizations. For more information, see Appendix E, “Creating Custom Data Visualizations” in the *MULTI: Debugging* book.

The following list provides a brief description of each data visualization command. For a command's arguments and for more information, see the page referenced.

- **dataview** — Opens a **Graph View** window displaying the specified profile or view (see “dataview” on page 306).
- **dvclear** — Clears all loaded data descriptions (see “dvclear” on page 306).
- **dvload** — Loads the specified **.mdv** file, which is a custom data visualization describing the data definitions, profiles, and views to be used for your MULTI session (see “dvload” on page 307).
- **dvprofile** — Makes the specified profile the active profile (see “dvprofile” on page 307).

dataview

dataview [*profname* | *view_name*]

GUI only

Opens a **Graph View** window displaying the specified profile (*profname*) or view (*view_name*). The graph will use the default root (*defroot*) specified in the profile or view definition.

If no argument is specified, all views will be displayed.

dvclear

dvclear

GUI only

Clears all loaded data descriptions.

dvload

dvload *filename*

GUI only

Loads the specified file of data descriptions, where *filename* is the name of a custom data visualization (**.mdv**) describing the data definitions, profiles, and views to be used for your MULTI session.

dvprofile

dvprofile *profname*

GUI only

Makes the profile with the profile name *profname* the active profile. (See “Profile Descriptions” in Appendix E, “Creating Custom Data Visualizations” in the *MULTI: Debugging* book.)

Appendix A

Deprecated Command Reference

Contents

Deprecated Commands	310
---------------------------	-----

The commands in this appendix have been deprecated in MULTI and will be removed from a future release.

Deprecated Commands

The following list identifies deprecated commands.

- **ba**
See “b” on page 46 instead.
- **br**
See “b” on page 46 instead.
- **bR**
See “bA” on page 49 instead.
- **browse procs | procedures**
See **browse funcs | functions** in “browse” on page 294 instead.
- **findleaks**
See “heapview” on page 297 instead.
- **infiniteview**
See “memview” on page 299 instead.
- **l p**
See **l f** in “l” on page 115 instead.
- **multibar**
See “launcher” on page 27 instead.
- **refresh**
See “load” on page 251 instead.
- **remote**
See “connect” on page 247 instead.
- **romverify**
See “verify” on page 143 instead.
- **setsync**
setsync has been renamed to **setgroupties**, which is deprecated. If you depend on this command, please contact Green Hills Support. For an alternative, see “syncrc” on page 185.

- **showsync**
showsync has been renamed to **showgroupties**, which is deprecated. If you depend on this command, please contact Green Hills Support. For an alternative, see “syncrc” on page 185.
- **tlist**
To open the Task Manager in a run-mode debugging environment, see “taskwindow” on page 230 instead. To open an **OSA Explorer** on the current process in a freeze-mode debugging environment or on the current debug server in a run-mode debugging environment, see “osaexplorer” on page 225.

Index

Symbols

- ! (exclamation point) command, 215
- !! (exclamation point-double) command, 215
- %w (percent w) key sequence, 150
- + (plus sign) command, 147
- (minus sign) command, 147
- > (minus sign, right angle bracket) command, 95
- / (slash) command, 237
- < (left angle bracket) command, 233
- > (right angle bracket) command, 232
- >> (right angle bracket-double) command, 232
- ? (question mark)
 - command, 237
- @bp_count argument, 4, 44
- @continue_count argument, 4, 162
- { } (curly braces)
 - indicating command lists with, 5, 12

A

- about command, 124
- About dialog box, 124
- aboutlic command, 125
- addhook command, 217
- address expressions, 5
 - toggling status of, 70
- addresses
 - halting process on write to with watchpoint, 71
- alertdialog command, 209
- alias command, 199
- asm command, 17
- _ASMCache system variable
 - toggling with caches, 20
- asmmode command, 108
- assem command, 108
- assembling
 - instructions with asm, 17
- attach command, 18
- attaching to a process, 18

B

- b command, 46
- B command, 48
- ba command (deprecated), 310
- bA command, 49
- backhistory command, 216
- bc command, 164
- bcB command, 164
- bcBhardBrk command, 165
- bchardBrk command, 165
- bcU command, 179
- bi command, 49
- bI command, 49
- bif command, 50
- blocking
 - commands, 164, 165, 166, 167, 168, 178
- %bp_ID argument, 3
- %bp_label argument, 4
- bpload command, 51
- bprev command, 180
- bpsave command, 51
- bpview command, 52
- br command (deprecated), 310
- bR command (deprecated), 310
- braces, curly ({})
 - indicating command lists with, 5, 12
- break command, 205
- breakpoints
 - commands for, 44
 - conditional, setting with bif, 50
 - continue count, specifying with @continue_count, 4, 162
 - count, specifying with @bp_count, 4, 44
 - deleting, 55, 56
 - exit point, setting, 54
 - IDs, 3, 10, 11
 - information about, 48, 56
 - installing on request, 64
 - on instructions, setting, 49
 - labels, 4, 10, 11
 - lists, 11
 - loading with bpload, 51
 - ranges, 11
 - restoring deleted, 56
 - saving, 51
 - scripted, 5, 12, 44
 - setting basic, 46
 - temporary, setting with bA, 49
 - toggling status of, 70, 71
 - up-level, setting, 53
- breakpoints command, 52

Breakpoints window

opening, 52

browse command, 294

browse procedures command (deprecated), 310

browse procs command (deprecated), 310

Browse window

commands for, 294, 295

browseref command, 295

bs command, 180

bsearch command, 238

bsi command, 180

bt command, 52

bu command, 53

bU command, 53

bugreport command, 125

build command, 74

Builder

opening, 74

builder command, 74

building

commands, 74

buttons

commands for, 94, 95

configuring, 94, 95, 96

bx command, 54

bX command, 54

C

c command, 166

Cache Find window, 305

_CACHE system variable

toggling with caches, 20

Cache View window, 305

cachefind command, 305

caches

commands for viewing, 305

toggling _CACHE and _ASMCACHE, 20

caches command, 20

cacheview command, 305

call command, 20

call stack (see Call Stack window)

Call Stack window

commands, 78, 79, 81

configuring with cvconfig, 81

functions, listing, 79

calls command, 78

callsview command, 79

case sensitivity of searches

changing, 238

cat command, 109

cb command, 166

cbhardbrk command, 167

cedit command, 199

cf command, 167

cfb command, 168

change_binding command, 247

changegroup command, 264

chardbrk command, 168

chgcase command, 238

clear command, 109

clearconfig command, 85

clearhooks command, 219

closing multiple windows, 303

colors

for syntax, configuring, 93

comeback command, 109

command conventions, 3

command groups

breakpoint commands, 44

building commands, 74

button, menu, mouse commands, 94

cache view commands, 305

call stack commands, 78

command manipulation and Debugger macro commands, 198

conditional program execution commands, 204

configuration commands, 84

continue commands, 162

data visualization commands, 306

Debugger macro and command manipulation commands, 198

Debugger Note commands, 102

deprecated commands, 310

dialog commands, 208

display and print commands, 106

external tool commands, 211

general Debugger commands, 16

general program execution commands, 160

general view commands, 292

halt commands, 170

help and information commands, 124

history commands, 214

information and help commands, 124

INTEGRITY TimeMachine commands, 270

memory commands, 128

menu, mouse, button commands, 94

mouse, menu, button commands, 94

navigation commands, 146

Object Structure Awareness (OSA) commands, 223

playback and record commands, 230

print and display commands, 106

- profiling commands, 154
- program execution commands, 160, 171, 173, 174, 178, 180, 181, 182, 183
- record and playback commands, 230
- register commands, 190
- run commands, 171
- scripting commands, 198
- search commands, 236
- serial connection commands, 260
- signal commands, 176
- single-stepping commands, 177
- synchronous debugging commands, 185
- target connection commands, 246
- task execution commands, 187
- task group commands, 264
- TimeMachine commands, 271
- trace commands, 271
- tracepoint commands, 284
- view commands, 292
- command line options
 - I, 14
- command lists, 5, 12
- command manipulation
 - commands for, 198
- commands
 - ! (exclamation point), 215
 - !! (exclamation point-double), 215
 - + (plus sign), 147
 - (minus sign), 147
 - > (minus sign, right angle bracket), 95
 - / (slash), 237
 - < (left angle bracket), 233
 - > (right angle bracket), 232
 - >> (right angle bracket-double), 232
 - ? (question mark), 237
 - about, 124
 - aboutlic, 125
 - addhook, 217
 - alrtdialog, 209
 - alias, 199
 - asm, 17
 - asmmode, 108
 - assem, 108
 - attach, 18
 - b, 46
 - B, 48
 - ba (deprecated), 310
 - bA, 49
 - backhistory, 216
 - bc, 164
 - bcb, 164
 - bcbbhardbrk, 165
 - bchardbrk, 165
 - bcU, 179
 - bi, 49
 - bI, 49
 - bif, 50
 - bpload, 51
 - bprev, 180
 - bpsave, 51
 - bpview, 52
 - br (deprecated), 310
 - bR (deprecated), 310
 - break, 205
 - breakpoints, 52
 - browse, 294
 - browse procedures (deprecated), 310
 - browse procs (deprecated), 310
 - browseref, 295
 - bs, 180
 - bsearch, 238
 - bsi, 180
 - bt, 52
 - bu, 53
 - bU, 53
 - bugreport, 125
 - build, 74
 - builder, 74
 - bx, 54
 - bX, 54
 - c, 166
 - cachefind, 305
 - caches, 20
 - cacheview, 305
 - call, 20
 - calls, 78
 - callsview, 79
 - cat, 109
 - cb, 166
 - cbhardbrk, 167
 - cedit, 199
 - cf, 167
 - cfb, 168
 - change_binding, 247
 - changegroup, 264
 - chardbrk, 168
 - chgcas, 238
 - clear, 109
 - clearconfig, 85
 - clearhooks, 219
 - comeback, 109
 - comments in, 13

compare, 129
compareb, 129
completeselection, 239
components, 110
configoptions, 85
configure, 86
configurefile, 86
connect, 247
connectionview, 250
continue, 205
copy, 130
copyb, 130
creategroup, 266
cu, 180
cU, 180
customizemenus, 95
customizetoolbar, 95
cvconfig, 81
d, 55
D, 56
dataview, 306
dbnew, 20
dbprint, 110
debug, 21
debugbutton, 96
debugpane, 111
debugsnapshot, 21
default search path of, 14
define, 200
deprecated, 310
destroygroup, 267
detach, 23
dialog, 209
dialogsearch, 239
diff, 296
directorydialog, 210
disassemble, 131
disconnect, 250
do, 206
dumpfile, 111
dvclear, 306
dvload, 307
dvprofile, 307
dz, 56
E, 112
e, 147
eagleconfig, 23
echo, 113
edit, 296
editbutton, 96
edithwbp, 58
editswbp, 58
edittp, 284
editview, 297
endtime, 26
eval, 113
evaltosocket, 212
examine, 114
exclamation point (!), 215
exclamation point-double (!!), 215
filedialog, 210
fileextensions, 87
fill, 132
fillb, 132
find, 133
findb, 133
findleaks (deprecated), 310
flash, 134
fontsize, 87
for, 206
forwardhistory, 216
fsearch, 239
g, 160
getargs, 160
goaway, 114
grep, 240
groupaction, 267
H, 170
h, 217
halt, 170
hardbrk, 59
heapview, 297
help, 125
help regarding, 2
history, 26
if, 207
imagename, 88
indexnext, 148
indexprev, 149
infiniteview (deprecated), 310
info, 125
inspect, 98
install_bp_on_request, 64
iobuffer, 251
isearch, 241
isearchadd, 242
isearchreturn, 242
k, 171
keybind, 98, 150
l, 115
l p (deprecated), 310
launcher, 27

left angle bracket (<), 233
 Linux/Solaris only, 2, 5
 listgroup, 268
 listhooks, 220
 load, 251
 loadconfigfromfile, 88
 loadsym, 27
 localsview, 298
 macrotrace, 200
 make, 212
 map, 117
 memdump, 136
 memload, 137
 memread, 139
 memtest, 139
 memview, 299
 memwrite, 142
 menu, 99
 mev, 28
 minus sign (-), 147
 minus sign, right angle bracket (->), 95
 monitor, 29
 mouse, 99, 150
 mprintf, 117
 mrulist, 118
 mrv, 30
 multibar (deprecated), 310
 mute, 119
 n, 182
 new, 31
 ni, 183
 nl, 184
 notedel, 102
 noteedit, 103
 notelist, 103
 notestate, 104
 noteview, 104
 number, 149
 osacleanup, 224
 osacmd, 225
 osaexplorer, 225
 osaFillGuiWithObj, 226
 osainject, 227
 osasetup, 227
 osatask, 228
 osatasklist, 229
 osaview, 229
 output, 32
 overview of, 2
 P, 33
 p, 119
 passive, 285
 plus sign (+), 147
 prepare_target, 253
 print, 119
 printline, 119
 printphys, 120
 printsearch, 243
 printwindow, 120
 profile, 154
 pwd, 120
 python, 222
 pywin, 223
 q, 35
 Q, 121
 question mark (?), 237
 quit, 35
 quitall, 36
 r, 171
 R, 172
 rb, 173
 Rb, 173
 recording to playback files, 230, 231
 refresh (deprecated), 310
 regadd, 191
 regappend, 191
 regbasefile, 192
 regload, 192
 regtab, 192
 regunload, 194
 regvalload, 194
 regvalsave, 194
 regview, 195
 remote (deprecated), 310
 repeating, 215
 reset, 256
 restart, 173
 restore, 37
 resume, 174
 return, 201
 right angle bracket (>), 232
 right angle bracket-double (>>), 232
 rominithbp, 65
 romverify (deprecated), 310
 route, 201
 runbacktohardbrk, 168
 rundir, 174
 runtohardbrk, 169
 runtohere, 169
 s, 181
 S, 182
 save, 37

saveconfig, 88
saveconfigtofile, 89
savedebugpane, 121
sb, 66
sc, 201
scrollcommand, 149
serialconnect, 260
serialdisconnect, 261
set_runmode_partner, 256
setargs, 161
setbrk, 67
setgroupities (deprecated), 310
sethbp, 68
setintegritydir, 89
setsync (deprecated), 310
setup, 257
setuvelocitydir, 90
shell, 202
showdef, 299
showgroupities (deprecated), 311
showhistory, 300
showsync (deprecated), 311
si, 182
Si, 183
signal, 176
sl, 183
Sl, 184
slash (/), 237
socket, 212
Solaris/Linux only, 2, 5
source, 90
sourceroot, 91
star, 92
starttime, 38
stepinto, 184
stopif, 68
stopifi, 69
substitute, 203
switch, 151
syncolor, 93
syncrc, 185
target, 258
targetinput, 259
taskaction, 187
taskwindow, 230
timemachine, 272
tlist (deprecated), 311
tmhistory, 270
tog, 70
Tog, 71
top, 300

tpdel, 285
tpenable, 286
tplist, 286
tpprint, 287
tppurge, 287
tpreset, 288
tpset, 288
trace, 273
tracebrowse, 277
tracedata, 277
tracefunction, 278
traceline, 278
traceload, 278
tracemevsys, 279
tracempath, 280
tracepro, 280
tracesave, 280
tracesavetext, 281
tracesubfunction, 282
unalias, 204
unload, 259
unloadsym, 38
update, 301
uptosource, 151
usage conventions, 3
usage, 126
verify, 143
view, 301
viewdel, 303
viewlist, 303
wait, 38
watchpoint, 71
wgutils, 75
while, 208
window, 304
windowcopy, 122
windowpaste, 122
windowspaste, 122
xmit, 258
xmitio, 259
xref, 295
zignal, 176
comments
 in commands, 13
compare command, 129
compareb command, 129
completeselection command, 239
components command, 110
conditional breakpoints
 setting with bif, 50
conditional program execution commands, 204

- configoptions command, 85
- configuration commands, 84
- configuration options
 - continuePlaybackFileOnError, 231
 - procRelativeLines, 5, 7
- configure command, 86
- configurefile command, 86
- configuring
 - buttons, 94, 95, 96
 - menus, 94, 95
 - mouse buttons, 94
 - syntax colors, 93
 - toolbar, 95
- connect command, 247
- connections
 - target (see targets)
- connectionview command, 250
- context-sensitive help, 125
- continue command, 205
- continue commands, 162
- CONTINUECOUNT system variable, 162
- continuePlaybackFileOnError configuration option, 231
- conventions
 - command, 3
 - typographical, xviii
- copy command, 130
- copyb command, 130
- creategroup command, 266
- cu command, 180
- cU command, 180
- curly braces ({})
 - indicating command lists with, 5, 12
- customizemenus command, 95
- customizetoolbar command, 95
- customizing (see configuring)
- cvconfig command, 81

D

- d command, 55
- D command, 56
- data visualization
 - commands for, 306
- dataview command, 306
- dbnew command, 20
- dbprint command, 110
- .dbs setup scripts, 248
- debug command, 21
- debugbutton command, 96
- Debugger
 - command conventions, 3

- GUI mode, 2, 5
- non-GUI mode, 5
- passive mode, 285
- Debugger macros
 - commands for, 198
- Debugger modes (see Debugger, GUI mode and non-GUI mode)
- Debugger Notes
 - commands for, 102
- debugging
 - in passive mode, 285
- debugpane command, 111
- debugsnapshot command, 21
- define command, 200
- deprecated commands, 310
- destroygroup command, 267
- detach command, 23
- dialog command, 209
- dialog commands, 208
- dialogsearch command, 239
- diff command, 296
- directorydialog command, 210
- disassemble command, 131
- disconnect command, 250
- display commands, 106
- do command, 206
- document set, xvi, xvii
- dumpfile command, 111
- dvclear command, 306
- dvload command, 307
- dvprofile command, 307
- dz command, 56

E

- E command, 112
- e command, 147
- e frame_ command, 112
- eagleconfig command, 23
- echo command, 113
- edit command, 296
- editbutton command, 96
- edithwbp command, 58
- editswbp command, 58
- edittp command, 284
- editview command, 297
- endtime command, 26
- epilogue code
 - setting breakpoints in, 54
- eval command, 113
- evaltosocket command, 212

- examine command, 114
- exceptions
 - toggling status of, 70
- exclamation point (!) command, 215
- exclamation point-double (!!) command, 215
- execution (see programs)
- exit breakpoints
 - setting, 54
- expressions
 - placeholders for, 5
- external tool commands, 211
- e 0_, equivalent to E command, 112

F

- file-relative line numbers
 - interpreting line numbers as, 7
- File-Relative Mode, 5, 7
- filedialog command, 210
- fileextensions command, 87
- fill command, 132
- fillb command, 132
- find command, 133
- findb command, 133
- findleaks command (deprecated), 310
- flash command, 134
- fontsize command, 87
- for command, 206
- forwardhistory command, 216
- freeze-mode debugging
 - tasks, 225
- fsearch command, 239
- function-relative line numbers
 - interpreting line numbers as, 7
- Function-Relative Mode, 5, 7

G

- g command, 160
- gbugrpt utility, 125
- general program execution commands, 160
- general view commands, 292
- getargs command, 160
- goaway command, 114
- grep command, 240
- groupaction command, 267
- GUI mode, 2
 - commands, 5
 - starting Debugger in, 2
- GUI only label, 5

H

- H command, 170
- h command, 217
- halt command, 170
- halt commands, 170
- hardbrk command, 59
- heapview command, 297
- help and information commands, 124
- help command, 125
- history command, 26
- history commands, 214

I

- I command line option, 14
- IDs, breakpoint, 3, 10, 11
- if command, 207
- imagename command, 88
- indexnext command, 148
- indexprev command, 149
- infiniteview command (deprecated), 310
- info command, 125
- information and help commands, 124
- inspect command, 98
- install_bp_on_request command, 64
- installing breakpoints on request, 64
- instructions
 - assembling, 17
 - setting breakpoints on, 49
- INTEGRITY TimeMachine commands, 270
- INTEGRITY TimeMachine Debugger commands
 - bc, 164
 - bc b, 164
 - bc bhardbrk, 165
 - bc hardbrk, 165
 - bcU, 179
 - bprev, 180
 - bs, 180
 - bsi, 180
 - runbacktohardbrk, 168
- iobuffer command, 251
- isearch command, 241
- isearchadd command, 242
- isearchreturn command, 242

K

- k command, 171
- keybind command, 98
- keys
 - configuring, 94

L

- l command, 115
- lp command (deprecated), 310
- labels
 - breakpoint, 4, 10, 11
 - GUI only, 5
 - Linux/Solaris only, 5
- launcher command, 27
- left angle bracket (<) command, 233
- licenses
 - viewing information about, 125
- line numbers
 - interpreting as file- or function-relative, 7
- Linux/Solaris
 - commands, 2, 5
 - non-GUI mode, 5
- Linux/Solaris only labels, 5
- listgroup command, 268
- lsthooks command, 220
- listing
 - breakpoint information, 48, 56
- lists
 - breakpoint, 11
- load command, 251
- loadconfigfromfile command, 88
- loading breakpoints with bpload, 51
- loadsym command, 27
- localsview command, 298
- loops
 - breaking out of with break, 205
 - do command, 206
 - for command, 206
 - while command, 208

M

- macros (see Debugger macros)
- macrotrace command, 200
- make command, 212
- manipulating commands, 198
- map command, 117
- .mbs setup scripts, 248
- memdump command, 136
- memload command, 137
- memory
 - commands, 128
 - commands for viewing caches, 305
 - content, viewing, 302
- memread command, 139
- memtest command, 139
- memview command, 299

- memwrite command, 142
- menu command, 99
- menus
 - commands for configuring, 94, 95
- mev command, 28
- minus sign (-) command, 147
- minus sign, right angle bracket (->) command, 95
- monitor command, 29
- mouse buttons
 - commands for configuring, 94
- mouse command, 99, 150
- mprintf command, 117
- mrulist command, 118
- mrvc command, 30
- MULTI Integrated Development Environment (IDE)
 - document set, xvii
 - exiting, 36
 - viewing information about, 124
- multibar command (deprecated), 310
- mute command, 119

N

- n command, 182
- navigating
 - commands for, 146
- new command, 31
- ni command, 183
- nl command, 184
- non-GUI mode
 - commands, 5
- non-intrusive debugging
 - with passive mode, 285
- notedel command, 102
- noteedit command, 103
- notelist command, 103
- notes (see Debugger Notes)
- notestate command, 104
- noteview command, 104
- number command, 149
- number_, 5

O

- Object Structure Awareness (OSA)
 - commands, 223
- one-shot (temporary) breakpoints
 - setting with bA, 49
- online help
 - for commands, 2
- options (see configuration options)
- OSA Explorer

- opening, 225
- osacleanup command, 224
- osacmd command, 225
- osaexplorer command, 225
- osaFillGuiWithObj command, 226
- osainject command, 227
- osasetup command, 227
- osatask command, 228
- osatasklist command, 229
- osaview command, 229
- output command, 32

P

- P command, 33
- p command, 119
- passive command, 285
- passive mode, 285
- percent w (%w) key sequence, 150
- playback and record commands, 230
- plus sign (+) command, 147
- prepare_target command, 253
- print command, 119
- print commands, 106
- printing
 - commands for, 106
- printline command, 119
- printphys command, 120
- printsearch command, 243
- printwindow command, 120
- Process Viewer
 - opening, 300
- processes
 - attaching to, 18
 - continuing stopped, 167, 168
 - halting, 71, 170
- procRelativeLines configuration option, 5, 7
- profile command, 154
- profiling
 - commands, 154
- programs
 - execution of, 160
 - (see also processes)
- pwd command, 120
- python command, 222
- pywin command, 223

Q

- q command, 35
- Q command, 121
- question mark (?) command, 237

- quit command, 35
- quitall command, 36

R

- r command, 171
- R command, 172
- ranges
 - breakpoint, 11
- rb command, 173
- Rb command, 173
- re-executing commands, 215
- record and playback commands, 230
- refresh command (deprecated), 310
- regadd command, 191
- regappend command, 191
- regbasefile command, 192
- register commands, 190
- regload command, 192
- regtab command, 192
- regunload command, 194
- regvalload command, 194
- regvalsave command, 194
- regview command, 195
- remote command (deprecated), 310
- repeating commands, 215
- reset command, 256
- restart command, 173
- restore command, 37
- resume command, 174
- return command, 201
- right angle bracket (>) command, 232
- right angle bracket-double (>>) command, 232
- rominithbp command, 65
- romverify command (deprecated), 310
- route command, 201
- run commands, 171
- run-mode debugging
 - tasks, 230
- runbacktohardbrk command, 168
- rundir command, 174
- runtohardbrk command, 169
- runtohere command, 169

S

- s command, 181
- S command, 182
- S-Record format, 136, 137
- save command, 37
- saveconfig command, 88
- saveconfigtofile command, 89

savedebugpane command, 121
 sb command, 66
 sc command, 201
 scripted breakpoints, 5, 12, 44
 scripts
 commands for, 198
 .dbs, 248
 .mbs, 248
 scrollcommand command, 149
 searching
 case sensitivity, changing, 238
 commands for, 236
 default path for, 14
 serial connections
 commands for, 260
 serialconnect command, 260
 serialdisconnect command, 261
 set_runmode_partner command, 256
 setargs command, 161
 setbrk command, 67
 setgrouppties command (deprecated), 310
 sethbp command, 68
 setintegritydir command, 89
 setsync command (deprecated), 310
 setup command, 257
 setuvelocitydir command, 90
 shell command, 202
 showdef command, 299
 showgrouppties command (deprecated), 311
 showhistory command, 300
 showsync command (deprecated), 311
 si command, 182
 Si command, 183
 signal command, 176
 signals
 commands relating to, 176
 single-stepping
 commands for, 177
 sl command, 183
 Sl command, 184
 slash (/) command, 237
 socket command, 212
 Solaris/Linux
 commands, 2, 5
 non-GUI mode, 5
 source command, 90
 sourceroot command, 91
 stack trace commands (see Call Stack window, commands)
 stacklevel_, 5
 star command, 92
 starting

Debugger
 in GUI mode, 2
 starttime command, 38
 stepinto command, 184
 stopif command, 68
 stopifi command, 69
 stopped process, continuing, 167, 168
 substitute command, 203
 switch command, 151
 synchronous debugging commands, 185
 syncolor command, 93
 syncrc command, 185
 syntax coloring
 configuring, 93

T

target command, 258
 targetinput command, 259
 targets
 commands relating to, 246
 task group commands, 264
 Task Manager
 opening, 230
 taskaction command, 187
 tasks
 commands relating to, 187
 taskwindow command, 230
 temporary breakpoints
 setting with bA, 49
 timemachine command, 272
 TimeMachine commands, 271
 TimeMachine Debugger commands
 bc, 164
 bcb, 164
 bcbhardbrk, 165
 bchardbrk, 165
 bcU, 179
 bprev, 180
 bs, 180
 bsi, 180
 runbacktohardbrk, 168
 tlist command (deprecated), 311
 tmhistory command, 270
 tog command, 70
 Tog command, 71
 toggling
 address expression status, 70
 breakpoint status, 70, 71
 exception status, 70
 toolbar, Debugger

- configuring, 95
- top command, 300
- tpdel command, 285
- tpenable command, 286
- tplist command, 286
- tpprint command, 287
- tppurge command, 287
- tpreset command, 288
- tpset command, 288
- trace
 - commands for collecting and using, 271
- trace command, 273
- tracebrowse command, 277
- tracedata command, 277
- tracefunction command, 278
- traceline command, 278
- traceload command, 278
- tracemevsys command, 279
- tracepath command, 280
- tracepoints
 - buffer
 - purging, 287
 - viewing, 287
 - commands, 284
 - timeout feature, 288
- tracepro command, 280
- tracesave command, 280
- tracesavetext command, 281
- tracesubfunction command, 282
- tracing program execution, 52, 271
 - (see also TimeMachine Debugger)
 - (see also trace)
- Tree Browser
 - command for, 294
- typographical conventions, xviii

U

- unalias command, 204
- unload command, 259
- unloadsym command, 38
- up-level breakpoints
 - setting, 53
- update command, 301
- uptosource command, 151
- usage command, 126
- usage for commands, 2
- Utility Program Launcher, 75

V

- verify command, 143

- view command, 301
- view commands, 292
- viewdel command, 303
- viewing
 - information about licenses, 125
 - information about MULTI, 124
- viewlist command, 303

W

- wait command, 38
- watchpoint command, 71
- watchpoints
 - setting with watchpoint, 71
- wgutils command, 75
- while command, 208
- window command, 304
- windowcopy command, 122
- windowpaste command, 122
- windows
 - closing multiple, 303
 - identification numbers for, 150
- windowspaste command, 122

X

- xmit command, 258
- xmitio command, 259
- xref command, 295

Z

- signal command, 176