Debugging with GDB

The GNU Source-Level Debugger

HP Eighteenth Edition, for GDB
February 2008

Richard Stallman, Roland Pesch, Stan Shebs, et al.

(Send bugs and comments on GDB to bug-gdb@gnu.org with copy to wdb-help@cup.hp.com)

Debugging with GDB
TEXinfo 2003-02-03.16

Published by the Free Software Foundation
59 Temple Place - Suite 330,
Boston, MA 02111-1307 USA
ISBN 1-882114-77-9

Table of Contents

Summary of GDB 1
Free software 1
Contributors to GDB 1

1 A Sample GDB Session 5
1.1 Loading the Executable 5
1.2 Setting Display width 6
1.3 Setting Breakpoints 6
1.4 Running the executable under GDB 6
1.5 Stepping to the next line in the source program 6
1.6 Stepping into a subroutine 6
1.7 Examining the Stack 7
1.8 Printing Variable Values 7
1.9 Listing Source Code 7
1.10 Setting Variable Values During a Session 8

2 Getting In and Out of GDB 11
2.1 Invoking GDB 11
2.1.1 Choosing files 11
2.1.2 Choosing modes 13
2.1.3 Redirecting WDB input and output to a file 15
2.2 Quitting GDB 15
2.3 Shell commands 16

3 GDB Commands 17
3.1 Command syntax 17
3.2 Command completion 17
3.3 Getting help 19

4 Running Programs Under GDB 23
4.1 Compiling for debugging 23
4.2 Starting your program 23
4.3 Arguments To Your Program 24
4.4 Program Environment 25
4.5 Working directory 26
4.6 Program Input and Output 26
4.7 Debugging a Running Process 27
4.8 Killing the child process 28
4.9 Debugging programs with multiple threads 28
4.10 Debugging programs with multiple processes 31

5 Stopping and Continuing 33

5.1 Breakpoints 33
5.1.1 Setting breakpoints 33
5.1.2 Setting catchpoints 37
5.1.3 Deleting breakpoints 38
5.1.4 Disabling breakpoints 39
5.1.5 Break conditions 40
5.1.6 Breakpoint command lists 41
5.1.7 Breakpoint menus 42
5.1.8 “Cannot insert breakpoints” 43
5.2 Continuing and stepping 43
5.3 Signals 46
5.4 Stopping and starting multi-thread programs 48

6 Examining the Stack 51

6.1 Stack frames 51
6.2 Stacks Without frames 51
6.3 Commands for Examining the Stack 52
6.4 Backtraces 52
6.5 Selecting a frame 53
6.6 Information about a frame 54

7 Examining Source Files 57

7.1 Printing source lines 57
7.2 Searching source files 58
7.3 Specifying source directories 59
7.4 Source and machine code 59

8 Examining Data 63

8.1 Expressions 63
8.2 Program variables 64
8.3 Artificial arrays 65
8.4 Output formats 66
8.5 Examining memory 67
8.6 Automatic display 68
8.7 Print settings 70
8.8 Value history 74
8.9 Convenience variables 75
8.10 Registers 76
8.11 Printing Floating Point Values 78
8.12 Floating point hardware 78

9 Using GDB with Different Languages 79
9.1 Switching between source languages 79
9.1.1 List of filename extensions and languages 79
9.1.2 Setting the working language 80
9.1.3 Having GDB infer the source language 80
9.2 Displaying the language 80
9.3 Type and range checking 81
9.3.1 An overview of type checking 81
9.3.2 An overview of range checking 82
9.4 Supported languages 83
9.4.1 C and C++ 83
9.4.1.1 C and C++ operators 84
9.4.1.2 C and C++ constants 85
9.4.1.3 C++ expressions 86
9.4.1.4 C and C++ defaults 87
9.4.1.5 C and C++ type and range checks 88
9.4.1.6 GDB and C 88
9.4.1.7 GDB features for C++ 88
9.4.2 Fortran 89
9.4.2.1 Fortran types 90
9.4.2.2 Fortran operators 90
9.4.2.3 Fortran special issues 91

10 Examining the Symbol Table 93

11 Altering Execution 97
11.1 Assignment to variables 97
11.2 Continuing at a different address 98
11.3 Giving your program a signal 99
11.4 Returning from a function 99
11.5 Calling program functions 100
11.6 Patching programs 100

12 GDB Files 103
12.1 Commands to specify files 103
12.2 Specifying shared library locations 106
12.3 Errors reading symbol files 107

13 Specifying a Debugging Target 109
13.1 Active targets 109
13.2 Commands for managing targets 109
13.3 Choosing target byte order 111

14 HP-UX Configuration-Specific Information . . . 113
14.1 Summary of HP Enhancements to GDB . . . 113
14.2 HP-UX dependencies . . . 116
14.2.1 Linker Dependencies . . . 116
14.2.2 Dependent Standard Library Routines for Run Time Checking . . . 116
14.3 Supported Platforms and Modes . . . 117
14.4 HP-UX targets . . . 118
14.5 Support for Alternate root . . . 118
14.6 Specifying object file directories . . . 119
14.7 Fix and continue debugging . . . 120
14.7.1 Fix and Continue compiler dependencies . . . 121
14.7.2 Fix and Continue restrictions . . . 121
14.7.3 Using Fix and Continue . . . 122
14.7.4 Example Fix and Continue session . . . 122
14.8 Inline Support . . . 124
14.8.1 Inline Debugging in HP 9000 Systems . . . 125
14.8.2 Inline Debugging in Integrity Systems . . . 125
14.8.2.1 Debugging Inline Functions in Integrity Systems . . . 127
14.9 Debugging Macros . . . 128
14.9.1 Viewing and Evaluating Macro Definitions . . . 128
14.9.1.1 Compiler Options to Enable Macro Debugging . . . 129
14.9.2 Examples for Macro Debugging . . . 129
14.10 Debugging Memory Problems . . . 133
14.10.1 When to suspect a memory leak . . . 133
14.10.2 Memory debugging restrictions . . . 133
14.10.3 Memory Debugging Methodologies . . . 133
14.10.4 Debugging Memory in Interactive Mode . . . 134
14.10.4.1 Commands for interactive memory debugging . . . 134
14.10.4.2 Example for interactive debugging session . . . 137
14.10.5 Debugging Memory in Batch Mode . . . 138
14.10.5.1 Setting Configuration Options for Batch Mode . . . 138
14.10.5.2 Environment variable setting for Batch mode debugging . . . 141
14.10.5.3 Example for Batch Mode RTC . . . 142
14.10.6 Debugging Memory Interactively After Attaching to a Running Process . . . 144
14.10.7 Configuring memory debugging settings . . . 145
14.10.7.1 Specifying the stack depth . . . 146
14.10.7.2 Specifying minimum leak size . . . 146
14.10.7.3 Specifying minimum block size . . . 146
14.10.8 Scenarios in memory debugging . . . 147

14.10.8.1 Stop when freeing unallocated or deallocated blocks 147
14.10.8.2 Stop when freeing a block if bad writes occurred outside block boundary 147
14.10.8.3 Stop when a specified block address is allocated or deallocated 147
14.10.8.4 Scramble previous memory contents at malloc/free calls 148
14.10.8.5 Detect dangling pointers and dangling blocks 148
14.10.8.6 Detect in-block corruption of freed blocks 149
14.10.8.7 Specify the amount of guard bytes for every block of allocated memory 149
14.10.9 Comparison of Memory Debugging Commands in Interactive Mode and Batch Mode 149
14.10.10 Heap Profiling 152
14.10.10.1 Commands for heap profiling 152
14.10.10.2 `info heap arena` 152
14.10.10.3 `info heap arena [0 |1|2|..] blocks stacks` 152
14.10.10.4 `info module ADDRESS` 153
14.10.10.5 `info heap process` 153
14.10.10.6 Example for heap profiling 153
14.10.11 Memory Checking Analysis for User Defined Memory Management Routines 154
14.10.12 Commands to track the change in data segment value 154
14.11 Thread Debugging Support 155
14.11.1 Support for Enabling and Disabling Specific Threads 155
14.11.2 Backtrace Support for Thread Debugging 156
14.11.3 Advanced Thread Debugging Support 156
14.11.3.1 Pre-requisites for Advanced Thread Debugging 157
14.11.3.2 Enabling and Disabling Advanced Thread Debugging Features 157
14.11.3.3 Commands to view information on pthread primitives 160
14.11.4 Debugging Threads Interactively After Attaching to a Process 160
14.11.5 Thread Debugging in Batch Mode 161
14.11.5.1 Pre-requisites for Batch mode of Thread Debugging 161
14.11.5.2 Limitations in Batch mode of thread debugging 163
14.11.6 Known issues with Thread Debugging for Interactive and Batch mode 163

14.12 Debugging MPI Programs 163
14.13 Debugging multiple processes (programs with `fork` and `vfork` calls) 164
14.13.1 Ask mode for `set follow-fork-mode` 164
14.13.2 serial mode for `set follow-fork-mode` 164
14.13.3 Support for showing unwind info 164
14.13.4 Printing CFM and PFS registers 165
14.14 Debugging Core Files 165
14.14.1 Generating core files with `packcore /unpackcore/getcore` 165
14.14.2 Support for the `dumpcore` command 166
14.14.2.1 Enhancements to the dumpcore command 167
14.14.3 Support for display of run time type information 167
14.15 Printing the Execution Path Entries for the Current Frame or Thread 167
14.15.1 Compiler Dependencies for Printing the Execution Path Entries 168
14.15.2 Example Illustrating Execution Path Recovery 169
14.16 Invoking GDB Before a Program Aborts 171
14.17 Aborting a Command Line Call 171
14.18 Instruction Level Stepping 172
14.19 Enhanced support for watchpoints and breakpoints 172
14.19.1 Deferred watchpoints 172
14.19.2 Hardware watchpoints 173
14.19.3 Hardware breakpoints 173
14.19.3.1 Setting breakpoints in unstripped shared library 173
14.19.4 Support for procedural breakpoints 173
14.19.5 Support for template breakpoints 174
14.20 Debugging support for shared libraries 174
14.20.1 Using shared library as main program 174
14.20.2 Setting Deferred Breakpoints in Shared Library 175
14.20.3 Using `catch load` 175
14.20.4 Privately mapping shared libraries 175
14.20.5 Selectively Mapping Shared Libraries As Private 176
14.20.6 Setting breakpoints in shared library 177
14.21 Language support 177
14.21.1 Enhanced Java Debugging Support 177
14.21.2 Commands for Examining Java Virtual Machine(JVM) internals 177
14.21.2.1 Java subcommands 177
14.21.3 Support for stack traces in Java, C, and C++ programs 178

14.21.4 Support for 64-bit Java, C, aC++ stack unwinding 178
14.21.5 Enhanced support for C++ templates 179
14.21.6 Support for `__fpreg` data type on IPF 180
14.21.7 Support for _Complex variables in HP C 180
14.21.8 Support for debugging namespaces 180
14.21.9 Command for evaluating the address of an expression 181
14.22 Viewing Wide Character Strings 181
14.23 Support for output logging 181
14.23.1 Support for dumping array in an ASCII file 182
14.23.2 Support for Fortran array slices 182
14.23.3 Displaying enumerators 183
14.23.4 Support for debugging typedefs 183
14.23.5 Support for `steplast` command for C and C++ 183
14.24 Getting information from a non-debug executable 184
14.25 Debugging optimized code 184
14.25.1 Debugging Optimized Code at Various Optimization Levels 186
14.25.1.1 `+O0` and `+O1` 186
14.25.1.2 `+O2/+O3/+O4/-ipo` 187
14.26 Visual Interface for WDB 188
14.26.1 Starting and stopping Visual Interface for WDB 189
14.26.2 Navigating the Visual Interface for WDB display 189
14.26.3 Specifying foreground and background colors 190
14.26.4 Using the X-window graphical interface 191
14.26.5 Using the TUI mode 191
14.26.6 Changing the size of the source or debugger pane 192
14.26.7 Using commands to browse through source files 192
14.26.8 Loading source files 193
14.26.9 Editing source files 193
14.26.10 Editing the command line and command-line history 193
14.26.11 Saving the contents of a debugging session to a file 193
14.27 Support for ddd 194
14.28 Support for XDB commands 194
14.28.1 stop in/at dbx commands 194
14.29 GNU GDB Logging Commands 194
14.30 Support for command line calls in a stripped executable 194
14.30.1 Support for command line calls in a stripped executable on PA-RISC systems 194

14.30.2 Additional support for command line calls in a stripped executable 195
14.30.2.1 For 32-bit applications: 195
14.30.2.2 For 64-bit applications 195
14.30.3 Support for debugging stripped binaries 196
14.30.3.1 Printing of locals and globals in a stripped module 196
14.30.3.2 Backtrace on stripped frames 196
14.30.3.3 Command line calls to non-stripped library 196
14.30.3.4 Setting breakpoints in unstripped shared library 196
14.31 Displaying the current block scope information 196
14.32 Linux support 197

15 The HP-UX Terminal User Interface 199

15.1 Starting the TUI 199
15.2 Automatically running a program at startup 200
15.3 Screen Layouts 201
15.3.1 Source pane 201
15.3.2 Disassembly pane 201
15.3.3 Source/Disassembly pane 202
15.3.4 Disassembly/Register pane 203
15.3.5 Source/Register pane 204
15.4 Cycling through the panes 205
15.5 Changing pane focus 205
15.6 Scrolling panes 208
15.7 Changing the register display 208
15.8 Changing the pane size 210
15.9 Refreshing and updating the window 211

16 XDB to WDB Transition Guide 213

16.1 By-function lists of XDB commands and HP WDB equivalents 213
16.1.1 Invocation commands 214
16.1.2 Window mode commands 214
16.1.3 File viewing commands 215
16.1.4 Source directory mapping commands 216
16.1.5 Data Viewing and modification commands 217
16.1.6 Stack viewing commands 219
16.1.7 Status-viewing command 219
16.1.8 Job control commands 220
16.2 Overall breakpoint commands 221
16.2.1 Auxiliary breakpoint commands 221
16.2.2 Breakpoint creation commands 222
16.2.3 Breakpoint status commands 223
16.2.4 All-procedures breakpoint commands 224
16.2.5 Global breakpoint commands 225

16.2.6 Assertion control commands 225
16.2.7 Record and playback commands 225
16.2.8 Macro facility commands 226
16.2.9 Signal control commands 226
16.2.10 Miscellaneous commands 227
16.3 XDB data formats and HP WDB equivalents 228
16.4 XDB location syntax and HP WDB equivalents 230
16.5 XDB special language operators and HP WDB equivalents 231
16.6 XDB special variables and HP WDB equivalents 231
16.7 XDB variable identifiers and HP WDB equivalents 233
16.8 Alphabetical lists of XDB commands and HP WDB equivalents 233
16.8.1 A 234
16.8.2 B 234
16.8.3 C through D 236
16.8.4 F through K 237
16.8.5 L 238
16.8.6 M through P 239
16.8.7 Q through S 240
16.8.8 T 242
16.8.9 U through Z 242
16.8.10 Symbols 244

17 Controlling GDB 249

17.1 Setting the GDB Prompt 249
17.2 Setting Command Editing Options in GDB 249
17.3 Setting Command History Feature in GDB 249
17.4 Setting the GDB Screen Size 251
17.5 Supported Number Formats 251
17.6 Optional warnings and messages 252
17.7 Optional messages about internal happenings 253

18 Canned Sequences of Commands 255

18.1 User-defined commands 255
18.2 User-defined command hooks 256
18.3 Command files 257
18.4 Commands for controlled output 257

19 Using GDB under GNU Emacs 259

20 GDB Annotations 261

20.1 What is an annotation? 261
20.2 The server prefix 261
20.3 Values 262
20.4 Frames 263
20.5 Displays 264
20.6 Annotation for GDB input 265
20.7 Errors 265
20.8 Information on breakpoints 266
20.9 Invalidation notices 266
20.10 Running the program 267
20.11 Displaying source 267
20.12 Annotations We Might Want in the Future 268

21 The GDB/MI Interface 269

Function and purpose 269
Notation and terminology 269
21.1 GDB/MI Command Syntax 269
21.1.1 GDB/MI Input syntax 269
21.1.2 GDB/MI Output syntax 270
21.1.3 Simple examples of GDB/MI interaction 272
21.2 GDB/MI compatibility with CLI 272
21.3 GDB/MI output records 273
21.3.1 GDB/MI result records 273
21.3.2 GDB/MI stream records 273
21.3.3 GDB/MI out-of-band records 273
21.4 GDB/MI command description format 274
21.5 GDB/MI breakpoint table commands 275
21.6 GDB/MI Data manipulation 283
21.7 GDB/MI program control 293
21.8 Miscellaneous GDB commands in GDB/MI 304
21.9 GDB/MI Stack Manipulation Commands 306
21.10 GDB/MI Symbol query commands 311
21.11 GDB/MI Target Manipulation Commands 314
21.12 GDB/MI thread commands 319
21.13 GDB/MI tracepoint commands 321
21.14 GDB/MI variable objects 321

22 Reporting Bugs in GDB 327

22.1 Have you found a bug? 327
22.2 How to report bugs 327

Appendix A Installing GDB 331

A.1 Compiling GDB in another directory 332
A.2 Specifying names for hosts and targets 333
A.3 `configure` options 334

Index **337**

Summary of GDB

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes—or what another program was doing at the moment it crashed.

GDB allows you to do the following:

- Load the executable along with any required arguments.
- Stop your program on specified blocks of code.
- Examine your program when it has stopped running due to an error.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C, C++ and Fortran. For more information, refer to the Section 9.4 [Supported languages], page 83. For more information on supported languages, refer to the Section 9.4.1 [C and C++], page 84.

GDB can be used to debug programs written in Fortran, although it may be necessary to refer to some variables with a trailing underscore. See Section 9.4.2 [Fortran], page 90.

This version of the manual documents WDB, implemented on HP 9000 or HP Integrity systems running Release 11.x of the HP-UX operating system. WDB can be used to debug code generated by the HP ANSI C, HP ANSI aC++ and HP Fortran compilers as well as the GNU C and C++ compilers. It does not support the debugging of Pascal, Modula-2 or Chill programs.

Free software

GDB is *free software*, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

Contributors to GDB

Richard Stallman was the original author of GDB, and of many other GNU programs. Many others have contributed to its development. This section attempts to credit major contributors. One of the virtues of free software is that everyone is free to contribute to it; with regret, we cannot actually acknowledge everyone here. The file '`ChangeLog`' in the GDB distribution approximates a blow-by-blow account.

Changes much prior to version 2.0 are lost in the mists of time.

> *Plea:* Additions to this section are particularly welcome. If you or your friends (or enemies, to be evenhanded) have been unfairly omitted from this list, we would like to add your names!

So that they may not regard their many labors as thankless, we particularly thank those who shepherded GDB through major releases: Andrew Cagney (release 5.0); Jim Blandy (release 4.18); Jason Molenda (release 4.17); Stan Shebs (release 4.14); Fred Fish (releases 4.16, 4.15, 4.13, 4.12, 4.11, 4.10, and 4.9); Stu Grossman and John Gilmore (releases 4.8, 4.7, 4.6, 4.5, and 4.4); John Gilmore (releases 4.3, 4.2, 4.1, 4.0, and 3.9); Jim Kingdon (releases 3.5, 3.4, and 3.3); and Randy Smith (releases 3.2, 3.1, and 3.0).

Richard Stallman, assisted at various times by Peter TerMaat, Chris Hanson, and Richard Mlynarik, handled releases through 2.8.

Michael Tiemann is the author of most of the GNU C++ support in GDB, with significant additional contributions from Per Bothner. James Clark wrote the GNU C++ demangler. Early work on C++ was by Peter TerMaat (who also did much general update work leading to release 3.0).

GDB 4 uses the BFD subroutine library to examine multiple object-file formats; BFD was a joint project of David V. Henkel-Wallace, Rich Pixley, Steve Chamberlain, and John Gilmore.

David Johnson wrote the original COFF support; Pace Willison did the original support for encapsulated COFF.

Brent Benson of Harris Computer Systems contributed DWARF 2 support.

Adam de Boor and Bradley Davis contributed the ISI Optimum V support. Per Bothner, Noboyuki Hikichi, and Alessandro Forin contributed MIPS support. Jean-Daniel Fekete contributed Sun 386i support. Chris Hanson improved the HP9000 support. Noboyuki Hikichi and Tomoyuki Hasei contributed Sony/News OS 3 support. David Johnson contributed Encore Umax support. Jyrki Kuoppala contributed Altos 3068 support. Jeff Law contributed HP PA and SOM support. Keith Packard contributed NS32K support. Doug Rabson contributed Acorn Risc Machine support. Bob Rusk contributed Harris Nighthawk CX-UX support. Chris Smith contributed Convex support (and Fortran debugging). Jonathan Stone contributed Pyramid support. Michael Tiemann contributed SPARC support. Tim Tucker contributed support for the Gould NP1 and Gould Powernode. Pace Willison contributed Intel 386 support. Jay Vosburgh contributed Symmetry support.

Andreas Schwab contributed M68K Linux support.

Rich Schaefer and Peter Schauer helped with support of SunOS shared libraries.

Jay Fenlason and Roland McGrath ensured that GDB and GAS agree about several machine instruction sets.

Patrick Duval, Ted Goldstein, Vikram Koka and Glenn Engel helped develop remote debugging. Intel Corporation, Wind River Systems, AMD, and ARM contributed remote debugging modules for the i960, VxWorks, A29K UDI, and RDI targets, respectively.

Brian Fox is the author of the readline libraries providing command-line editing and command history.

Andrew Beers of SUNY Buffalo wrote the language-switching code, the Modula-2 support, and contributed the Languages chapter of this manual.

Fred Fish wrote most of the support for Unix System Vr4. He also enhanced the command-completion support to cover C++ overloaded symbols.

Hitachi America, Ltd. sponsored the support for H8/300, H8/500, and Super-H processors.

NEC sponsored the support for the v850, Vr4xxx, and Vr5xxx processors.

Mitsubishi sponsored the support for D10V, D30V, and M32R/D processors.

Toshiba sponsored the support for the TX39 Mips processor.

Matsushita sponsored the support for the MN10200 and MN10300 processors.

Fujitsu sponsored the support for SPARClite and FR30 processors.

Kung Hsu, Jeff Law, and Rick Sladkey added support for hardware watchpoints.

Michael Snyder added support for tracepoints.

Stu Grossman wrote gdbserver.

Jim Kingdon, Peter Schauer, Ian Taylor, and Stu Grossman made nearly innumerable bug fixes and cleanups throughout GDB.

The following people at the Hewlett-Packard Company contributed support for the PA-RISC 2.0 architecture, HP-UX 10.20, 10.30, and 11.x (narrow mode), HP's implementation of kernel threads, HP's aC++ compiler, and the terminal user interface: Ben Krepp, Richard Title, John Bishop, Susan Macchia, Kathy Mann, Satish Pai, India Paul, Steve Rehrauer, and Elena Zannoni. Kim Haase, Rosario de la Torre, Alex McKale, Michael Coulter, Carl Burch, Bharath Chndramohan, Diwakar Nag, Muthuswami, Dennis Handly, Subash Babu and Dipshikha Basu provided HP-specific information in this manual.

Cygnus Solutions has sponsored GDB maintenance and much of its development since 1991. Cygnus engineers who have worked on GDB full time include Mark Alexander, Jim Blandy, Per Bothner, Kevin Buettner, Edith Epstein, Chris Faylor, Fred Fish, Martin Hunt, Jim Ingham, John Gilmore, Stu Grossman, Kung Hsu, Jim Kingdon, John Metzler, Fernando Nasser, Geoffrey Noer, Dawn Perchik, Rich Pixley, Zdenek Radouch, Keith Seitz, Stan Shebs, David Taylor, and Elena Zannoni. In addition, Dave Brolley, Ian Carmichael, Steve Chamberlain, Nick Clifton, JT Conklin, Stan Cox, DJ Delorie, Ulrich Drepper, Frank Eigler, Doug Evans, Sean Fagan, David Henkel-Wallace, Richard Henderson, Jeff Holcomb, Jeff Law, Jim Lemke, Tom Lord, Bob Manson, Michael Meissner, Jason Merrill, Catherine Moore, Drew Moseley, Ken Raeburn, Gavin Romig-Koch, Rob Savoye, Jamie Smith, Mike Stump, Ian Taylor, Angela Thomas, Michael Tiemann, Tom Tromey, Ron Unrau, Jim Wilson, and David Zuhn have made contributions both large and small.

1 A Sample GDB Session

This chapter describes the most common GDB commands with the help of an example.

The following topics are discussed:

- Loading the Executable
- Setting the Display Width
- Setting Breakpoints
- Running the Executable under GDB
- Stepping to the next line
- Stepping into a Subroutine
- Examining the Stack
- Printing Variable Values
- Listing the Source Code
- Setting Variable Values During a Debug Session

In this sample session, we emphasize user input like this: **input**, to make it easier to pick out from the surrounding output.

One of the preliminary versions of GNU `m4` (a generic macro processor) exhibits the following bug: sometimes, when we change its quote strings from the default, the commands used to capture one macro definition within another stop working. In the following short `m4` session, we define a macro `foo` which expands to `0000`; we then use the `m4` built-in `defn` to define `bar` as the same thing. However, when we change the open quote string to `<QUOTE>` and the close quote string to `<UNQUOTE>`, the same procedure fails to define a new synonym `baz`:

```
$ cd gnu/m4                    //change your current directory to the location where the m4 ex-
ecutable is stored.
$ ./m4                         //run the m4 application
define(foo,0000)

foo
0000
define(bar,defn(‘foo’))

bar
0000
changequote(<QUOTE>,<UNQUOTE>)

define(baz,defn(<QUOTE>foo<UNQUOTE>))
baz
C-d
m4: End of input: 0: fatal error: EOF in string
```

1.1 Loading the Executable

Let us use GDB to try to see what is going on.

```
$ (gdb) m4

HP gdb 3.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
```

```
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 3.0 (based on GDB ) is covered by the
GNU General Public License. Type "show copying" to see the conditions to
change it and/or distribute copies. Type "show warranty" for warranty/support.
```

GDB reads only enough symbol data to know where to find the rest when needed; as a result, the first prompt comes up very quickly.

1.2 Setting Display width

We now tell GDB to use a narrower display width than usual, so that examples fit in this manual.

```
((gdb)) set width 70
```

We need to see how the `m4` built-in `changequote` works. Having looked at the source, we know the relevant subroutine is `m4_changequote`, so we set a breakpoint there with the GDB `break` command.

1.3 Setting Breakpoints

Here we describe how to set a breakpoint.

```
((gdb)) break m4_changequote
Breakpoint 1 at 0x62f4: file builtin.c, line 879.
```

1.4 Running the executable under GDB

Using the `run` command, we start `m4` under GDB control. As long as the control does not reach the `m4_changequote` subroutine, the program runs as usual.

```
((gdb)) run
Starting program: /work/Editorial/gdb/gnu/m4/m4
define(foo,0000)

foo
0000
```

To trigger the breakpoint, we call `changequote`. GDB suspends execution of `m4`, displaying information about the context where it stops.

```
changequote(<QUOTE>,<UNQUOTE>)

Breakpoint 1, m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:879
879         if (bad_argc(TOKEN_DATA_TEXT(argv[0]),argc,1,3))
```

1.5 Stepping to the next line in the source program

Now we use the command `n` (`next`) to advance execution to the next line of the current function.

```
((gdb)) n
882         set_quotes((argc >= 2) ? TOKEN_DATA_TEXT(argv[1])\
 : nil,
```

1.6 Stepping into a subroutine

The `set_quotes` looks like a promising subroutine. We can go into it by using the command `s` (`step`) instead of `next`. `step` goes to the next line to be executed in *any* subroutine, so it steps into `set_quotes`.

```
((gdb)) s
set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
530         if (lquote != def_lquote)
```

1.7 Examining the Stack

The display that shows the subroutine where `m4` is now suspended (and its arguments) is called a stack frame display. It shows a summary of the stack. We can use the `backtrace` command (which can also be spelled `bt`), to see where we are in the stack as a whole: the `backtrace` command displays a stack frame for each active subroutine.

```
((gdb)) bt
#0  set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
#1  0x6344 in m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:882
#2  0x8174 in expand_macro (sym=0x33320) at macro.c:242
#3  0x7a88 in expand_token (obs=0x0, t=209696, td=0xf7fffa30)
    at macro.c:71
#4  0x79dc in expand_input () at macro.c:40
#5  0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

We step through a few more lines to see what happens. The first two times, we can use '`s`'; the next two times we use `n` to avoid falling into the `xstrdup` subroutine.

```
((gdb)) s
0x3b5c  532         if (rquote != def_rquote)
((gdb)) s
0x3b80  535         lquote = (lq == nil || *lq == '\0') ?  \
def_lquote : xstrdup(lq);
((gdb)) n
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup(rq);
((gdb)) n
538         len_lquote = strlen(rquote);
```

1.8 Printing Variable Values

The last line displayed looks a little odd in the listing above; we can examine the variables `lquote` and `rquote` to see if they are in fact the new left and right quotes we specified. We use the command `p` (`print`) to view their values.

```
((gdb)) p lquote
$1 = 0x35d40 "<QUOTE>"
((gdb)) p rquote
$2 = 0x35d50 "<UNQUOTE>"
```

1.9 Listing Source Code

`lquote` and `rquote` are indeed the new left and right quotes. To look at some context, we can display ten lines of source surrounding the current line with the `l` (`list`) command.

```
((gdb)) l
533             xfree(rquote);
534
535         lquote = (lq == nil || *lq == '\0') ? def_lquote\
 : xstrdup (lq);
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup (rq);
537
538         len_lquote = strlen(rquote);
539         len_rquote = strlen(lquote);
540     }
541
542     void
```

Let us step past the two lines that set `len_lquote` and `len_rquote`, and then examine the values of those variables.

```
((gdb)) n
539         len_rquote = strlen(lquote);
((gdb)) n
540     }
((gdb)) p len_lquote
$3 = 9
((gdb)) p len_rquote
$4 = 7
```

1.10 Setting Variable Values During a Session

That certainly looks wrong, assuming `len_lquote` and `len_rquote` are meant to be the lengths of `lquote` and `rquote` respectively. We can set them to better values using the `p` command, since it can print the value of any expression—and that expression can include subroutine calls and assignments.

```
((gdb)) p len_lquote=strlen(lquote)
$5 = 7
((gdb)) p len_rquote=strlen(rquote)
$6 = 9
```

Is that enough to fix the problem of using the new quotes with the `m4` built-in `defn`? We can allow `m4` to continue executing with the `c` (`continue`) command, and then try the example that caused trouble initially:

```
((gdb)) c
Continuing.

define(baz,defn(<QUOTE>foo<UNQUOTE>))

baz
0000
```

Success! The new quotes now work just as well as the default ones. The problem seems to have been just the two typos defining the wrong lengths. We allow `m4` to exit by giving it an EOF as input:

```
C-d
Program exited normally.
```

The message '`Program exited normally.`' is from GDB; it indicates `m4` has finished executing. We can end our GDB session with the GDB `quit` command.

```
((gdb)) quit
```

2 Getting In and Out of GDB

This chapter discusses how to start GDB, and exit out of it. The essentials are:

- type '`(gdb)`' to start GDB.
- type *`quit`* or *`C-d`* to exit.

2.1 Invoking GDB

Invoke GDB by running the program `(gdb)`. Once started, GDB reads commands from the terminal until you tell it to exit.

You can also run `(gdb)` with a variety of arguments and options, to specify more of your debugging environment at the outset.

The command-line options described here are designed to cover a variety of situations; in some environments, some of these options may effectively be unavailable.

The most usual way to start GDB is with one argument, specifying an executable program:

```
(gdb) program
```

You can also start with both an executable program and a core file specified:

```
(gdb) program core
```

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

```
(gdb) program 1234
```

would attach GDB to process `1234` (unless you also have a file named '`1234`'; GDB does check for a core file first).

Taking advantage of the second command-line argument requires a fairly complete operating system; when you use GDB as a remote debugger attached to a bare board, there may not be any notion of "process", and there is often no way to get a core dump. GDB will warn you if it is unable to attach or to read core dumps.

You can run `(gdb)` without printing the front material, which describes GDB's non-warranty, by specifying `-silent`:

```
gdb -silent
```

You can further control how GDB starts up by using command-line options. GDB itself can remind you of the options available.

Type

```
(gdb) -help
```

to display all available options and briefly describe their use ('`(gdb) -h`' is a shorter equivalent).

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

2.1.1 Choosing files

When GDB starts, it reads any arguments other than options as specifying an executable file and core file (or process ID). This is the same as if the arguments were specified by the '`-se`' and '`-c`' options respectively. (GDB reads the first argument that does not have an associated option flag as equivalent to the '`-se`' option followed by that argument; and the second argument that does not have an associated option flag, if any, as equivalent to the '`-c`' option followed by that argument.)

If GDB has not been configured to included core file support, such as for most embedded targets, then it will complain about a second argument and ignore it.

Many options have both long and short forms; both are shown in the following list. GDB also recognizes the long forms if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with '`--`' rather than '`-`', though we illustrate the more usual convention.)

`-symbols` *`file`*
`-s` *`file`*
: Read symbol table from file *file*.

`-exec` *`file`*
`-e` *`file`*
: Use file *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

`-se` *`file`*
: Read symbol table from file *file* and use it as the executable file.

`-core` *`file`*
`-c` *`file`*
: Use file *file* as a core dump to examine.

`-c` *`number`*
: Connect to process ID *number*, as with the `attach` command (unless there is a file in core-dump format named *number*, in which case '`-c`' specifies that file as a core dump to read).

`-command` *`file`*
`-x` *`file`*
: Execute GDB commands from file *file*. See Section 18.3 [Command files], page 257.

`-directory` *`directory`*
`-d` *`directory`*
: Add *directory* to the path to search for source files.

`-m`
`-mapped`
: *Warning: this option depends on operating system facilities that are not supported on all systems.*
If memory-mapped files are available on your system through the `mmap` system call, you can use this option to have GDB write the symbols from your program into a reusable file in the current directory. If the program you are debugging is called '`/tmp/fred`', the mapped symbol file is '`/tmp/fred.syms`'. Future GDB debugging sessions notice the presence of this file, and can quickly map in symbol information from it, rather than reading the symbol table from the executable program.

The '`.syms`' file is specific to the host machine where GDB is run. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

`-r`

`-readnow` Read each symbol file's entire symbol table immediately, rather than the default, which is to read it incrementally as it is needed. This makes startup slower, but makes future operations faster.

You typically combine the `-mapped` and `-readnow` options in order to build a '`.syms`' file that contains complete symbol information. (See Section 12.1 [Commands to specify files], page 103, for information on '`.syms`' files.) A simple GDB invocation to do nothing but build a '`.syms`' file for future use is:

```
gdb -batch -nx -mapped -readnow programname
```

2.1.2 Choosing modes

You can run GDB in various alternative modes—for example, in batch mode or quiet mode.

`-nx`

`-n` Do not execute commands found in any initialization files (normally called '`.gdbinit`', or '`gdb.ini`' on PCs). Normally, GDB executes the commands in these files after all the command options and arguments have been processed. See Section 18.3 [Command files], page 257.

`-quiet`

`-silent`

`-q` "Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

`-batch` Run in batch mode. Exit with status `0` after processing all the command files specified with '`-x`' (and all commands from initialization files, if not inhibited with '`-n`'). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.

Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message

```
Program exited normally.
```

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

`-nowindows`

`-nw` "No windows". If GDB comes with a graphical user interface (GUI) built in, then this option tells GDB to only use the command-line interface. If no GUI is available, this option has no effect.

`-windows`

`-w` If GDB includes a GUI, then this option requires it to be used if possible.

`-cd` *directory*
: Run GDB using *directory* as its working directory, instead of the current directory.

`-dbx`
: Support additional `dbx` commands, including:

- `use`
- `status` (in `dbx` mode, `status` has a different meaning than in default GDB mode.)
- `whereis`
- `func`
- `file`
- `assign`
- `call`
- `stop`

`-fullname`
`-f`
: GNU Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time your program stops). This recognizable format looks like two '`\032`' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two '`\032`' characters as a signal to display the source code for the frame.

`-epoch`
: The Epoch Emacs-GDB interface sets this option when it runs GDB as a subprocess. It tells GDB to modify its print routines so as to allow Epoch to display values of expressions in a separate window.

`-annotate` *level*
: This option sets the *annotation level* inside GDB. Its effect is identical to using '`set annotate` *level*' (see Chapter 20 [Annotations], page 261). Annotation level controls how much information does GDB print together with its prompt, values of expressions, source lines, and other types of output. Level 0 is the normal, level 1 is for use when GDB is run as a subprocess of GNU Emacs, level 2 is the maximum annotation suitable for programs that control GDB.

`-async`
: Use the asynchronous event loop for the command-line interface. GDB processes all events, such as user keyboard input, via a special event loop. This allows GDB to accept and process user commands in parallel with the debugged process being run[1], so you don't need to wait for control to return to GDB before you type the next command. (*Note:* as of version 5.0, the target side of the asynchronous operation is not yet in place, so '`-async`' does not work fully yet.)

  When the standard input is connected to a terminal device, GDB uses the asynchronous event loop by default, unless disabled by the '`-noasync`' option.

[1] GDB built with DJGPP tools for MS-DOS/MS-Windows supports this mode of operation, but the event loop is suspended when the debug target runs.

`-noasync` Disable the asynchronous event loop for the command-line interface.

`-baud` *`bps`*
`-b` *`bps`* Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

`-tty` *`device`*
`-t` *`device`*
Run using *device* for your program's standard input and output.

`-tui` Use a Terminal User Interface. For information, use your Web browser to read the file '`tui.html`', which is usually installed in the directory `/opt/langtools/wdb/doc` on HP-UX systems. Do not use this option if you run GDB from Emacs (see see Chapter 19 [Using GDB under GNU Emacs], page 259).

`-xdb` Run in XDB compatibility mode, allowing the use of certain XDB commands. For information, see the file '`xdb_trans.html`', which is usually installed in the directory `/opt/langtools/wdb/doc` on HP-UX systems.

`-interpreter` *`interp`*
Use the interpreter *interp* for interface with the controlling program or device. This option is meant to be set by programs which communicate with GDB using it as a back end. For example, '`--interpreter=mi`' causes GDB to use the *gdbmi interface* (see Chapter 21 [The GDB/MI Interface], page 269).

`-write` Open the executable and core files for both reading and writing. This is equivalent to the '`set write on`' command inside GDB (see Section 11.6 [Patching], page 100).

`-statistics`
This option causes GDB to print statistics about time and memory usage after it completes each command and returns to the prompt.

`-version` This option causes GDB to print its version number and no-warranty blurb, and exit.

`-pid` This option causes GDB to attach to a running process.

`-inline` This option causes the debugger to start with the inline debugging on.

`-src_no_g`
This option is used to set the limited source level debugging without compiling.

2.1.3 Redirecting WDB input and output to a file

To redirect WDB input and output to a file, use either of these commands to start the debugger:

```
$ script log1
$ gdb
```

or

```
$ gdb | tee log1
```

2.2 Quitting GDB

`quit` [*expression*]

`q`
: To exit GDB, use the `quit` command (abbreviated `q`), or type an end-of-file character (usually *`C-d`*). If you do not supply *expression*, GDB will terminate normally; otherwise it will terminate using the result of *expression* as the error code.

An interrupt (often *`C-c`*) does not exit from GDB, but rather terminates the action of any GDB command that is in progress and returns to GDB command level. It is safe to type the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.

You can use the `detach` command to release an attached process or device.

2.3 Shell commands

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend GDB; you can just use the `shell` command.

`shell` *command string*
: Invoke a standard shell to execute *command string*. If it exists, the environment variable `SHELL` determines which shell to run. Otherwise GDB uses the default shell (‘`/bin/sh`’ on Unix systems, ‘`COMMAND.COM`’ on MS-DOS, etc.).

The utility `make` is often needed in development environments. You do not have to use the `shell` command for this purpose in GDB:

`make` *make-args*
: Execute the `make` program with the specified arguments. This is equivalent to ‘`shell make` *make-args*’.

3 GDB Commands

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous; and you can repeat certain GDB commands by typing just ⟨RET⟩. You can also use the ⟨TAB⟩ key to get GDB to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility).

3.1 Command syntax

- A GDB command is a single line of input. There is no limit on how long it can be.
- It starts with a command name, and is followed by arguments whose meaning depends on the command name.
- GDB command names can be truncated if that abbreviation is unambiguous. The possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. You can test abbreviations by using them as arguments to the `help` command.
- A blank line as input to GDB (typing just ⟨RET⟩) means to repeat the previous command. Some commands (for example, `run`) do not repeat this way. These are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat. The `list` and `x` commands, when you repeat them with ⟨RET⟩, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.
- GDB can also use ⟨RET⟩ in another way: to partition lengthy output, in a way similar to the common utility `more` (see ⟨undefined⟩ [Screen size], page ⟨undefined⟩). Since it is easy to press one ⟨RET⟩ too many in this situation, GDB disables command repetition after any command that generates this sort of display.
- Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files (see Section 18.3 [Command files], page 257).

3.2 Command completion

GDB can fill in the rest of a word in a command for you, if there is only one possibility; it can also show you what the valid possibilities are for the next word in a command, at any time. This works for GDB commands, GDB subcommands, and the names of symbols in your program.

Press the ⟨TAB⟩ key whenever you want GDB to fill out the rest of a word. If there is only one possibility, GDB fills in the word, and waits for you to finish the command (or press ⟨RET⟩ to enter it). For example, if you type

```
((gdb)) info bre ⟨TAB⟩
```

GDB fills in the rest of the word '`breakpoints`', since that is the only `info` subcommand beginning with '`bre`':

```
((gdb)) info breakpoints
```

You can either press ⟨RET⟩ at this point, to run the `info breakpoints` command, or backspace and enter something else, if '`breakpoints`' does not look like the command you expected. (If you were sure you wanted `info breakpoints` in the first place, you might as well just type ⟨RET⟩ immediately after '`info bre`', to exploit command abbreviations rather than command completion.)

If there is more than one possibility for the next word when you press ⟨TAB⟩, GDB sounds a bell. You can either supply more characters and try again, or just press ⟨TAB⟩ a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with '`make_`', but when you type *`b make_`*⟨TAB⟩ GDB just sounds the bell. Typing ⟨TAB⟩ again displays all the function names in your program that begin with those characters, for example:

```
((gdb)) b make_ ⟨TAB⟩
```

GDB sounds bell; press ⟨TAB⟩ again, to see:

```
make_a_section_from_file     make_environ
make_abs_section             make_function_type
make_blockvector             make_pointer_type
make_cleanup                 make_reference_type
make_command                 make_symbol_completion_list
((gdb)) b make_
```

After displaying the available possibilities, GDB copies your partial input ('`b make_`' in the example) so you can finish the command.

If you just want to see the list of alternatives in the first place, you can press *`M-?`* rather than pressing ⟨TAB⟩ twice. *`M-?`* means ⟨META⟩ *`?`*. You can type this either by holding down a key designated as the ⟨META⟩ shift on your keyboard (if there is one) while typing *`?`*, or as ⟨ESC⟩ followed by *`?`*.

Sometimes the string you need, while logically a "word", may contain parentheses or other characters that GDB normally excludes from its notion of a word. To permit word completion to work in this situation, you may enclose words in `'` (single quote marks) in GDB commands.

The most likely situation where you might need this is in typing the name of a C++ function. This is because C++ allows function overloading (multiple definitions of the same function, distinguished by argument type). For example, when you want to set a breakpoint you may need to distinguish whether you mean the version of `name` that takes an `int` parameter, `name(int)`, or the version that takes a `float` parameter, `name(float)`. To use the word-completion facilities in this situation, type a single quote `'` at the beginning of the function name. This alerts GDB that it may need to consider more information than usual when you press ⟨TAB⟩ or *`M-?`* to request word completion:

```
((gdb)) b 'bubble( M-?
bubble(double,double)    bubble(int,int)
((gdb)) b 'bubble(
```

In some cases, GDB can tell that completing a name requires using quotes. When this happens, GDB inserts the quote for you (while completing as much as it can) if you do not type the quote in the first place:

```
((gdb)) b bub ⟨TAB⟩
```

GDB alters your input line to the following, and rings a bell:

```
    ((gdb)) b 'bubble(
```

In general, GDB can tell that a quote is needed (and inserts it) if you have not yet started typing the argument list when you ask for completion on an overloaded symbol.

For more information about overloaded functions, see Section 9.4.1.3 [C++ expressions], page 86. You can use the command `set overload-resolution off` to disable overload resolution; see Section 9.4.1.7 [GDB features for C++], page 88.

3.3 Getting help

You can always ask GDB itself for information on its commands, using the command `help`.

`help`
`h`
: You can use `help` (abbreviated `h`) with no arguments to display a short list of named classes of commands:

```
((gdb)) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without
               stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
((gdb))
```

`help` *`class`*
: Using one of the general help classes as an argument, you can get a list of the individual commands in that class. For example, here is the help display for the class `status`:

```
((gdb)) help status
Status inquiries.

List of commands:

info -- Generic command for showing things
        about the program being debugged
show -- Generic command for showing things
        about the debugger
```

```
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
((gdb))
```

`help` *`command`*
: With a command name as `help` argument, GDB displays a short paragraph on how to use that command.

`apropos` *`args`*
: The `apropos` *`args`* command searches through all of the GDB commands, and their documentation, for the regular expression specified in *args*. It prints out all matches found. For example:

```
apropos reload
```

results in:

```
set symbol-reloading -- Set dynamic symbol table reloading
                                 multiple times in one run
show symbol-reloading -- Show dynamic symbol table reloading
                                 multiple times in one run
```

`complete` *`args`*
: The `complete` *`args`* command lists all the possible completions for the beginning of a command. Use *args* to specify the beginning of the command you want completed. For example:

```
complete i
```

results in:

```
if
ignore
info
inspect
```

This is intended for use by GNU Emacs.

In addition to `help`, you can use the GDB commands `info` and `show` to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under `info` and under `show` in the Index point to all the sub-commands. See [Index], page 337.

`info`
: This command (abbreviated `i`) is for describing the state of your program. For example, you can list the arguments given to your program with `info args`, list the registers currently in use with `info registers`, or list the breakpoints you have set with `info breakpoints`. You can get a complete list of the `info` sub-commands with `help info`.

`set`
: You can assign the result of an expression to an environment variable with `set`. For example, you can set the GDB prompt to a $-sign with `set prompt $`.

`show`
: In contrast to `info`, `show` is for describing the state of GDB itself. You can change most of the things you can `show`, by using the related command `set`; for example, you can control what number system is used for displays with `set radix`, or simply inquire which is currently in use with `show radix`.

  To display all the settable parameters and their current values, you can use `show` with no arguments; you may also use `info set`. Both commands produce the same display.

Here are three miscellaneous `show` subcommands, all of which are exceptional in lacking corresponding `set` commands:

`show version`
: Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may need to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. Also, many system vendors ship variant versions of GDB, and there are variant versions of GDB in GNU/Linux distributions as well. The version number is the same as the one announced when you start GDB.

`show copying`
: Display information about permission for copying GDB.

`show warranty`
: Display the GNU "NO WARRANTY" statement, or a warranty, if your version of GDB comes with one.

4 Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it using compiler option `cc -g -O`.

You may start GDB with its arguments, if any, in an environment of your choice. If you are doing native debugging, you may redirect your program's input and output, debug an already running process, or kill a child process.

4.1 Compiling for debugging

Following points are noteable while compiling programs for debugging:

- Compile your program with the `-g-O` option to generate debugging information.
- The `-g-O` option is supported by HP ANSI C and HP aC++ compilers and GNU gcc compiler.
- Some compilers do not support the `-g-O` options together.
- The `-g-O` options do not work on machines with instruction scheduling.

  Note:

  Older versions of the GNU C compiler permitted a variant option '`-gg`' for debugging information. GDB no longer supports this format; if your GNU C compiler has this option, do not use it.

4.2 Starting your program

`run`
`r` Use the `run` command to start your program under GDB. You must first specify the program name (except on VxWorks) with an argument to GDB (see Chapter 2 [Getting In and Out of GDB], page 11), or by using the `file` or `exec-file` command (see Section 12.1 [Commands to specify files], page 103).

Note:

If you are running your program in an execution environment that supports processes, `run` creates an inferior process and makes that process run your program. (In environments without processes, `run` jumps to the start of your program.)

The execution of a program is affected by the information it receives from the parent process. You must provide GDB the information before starting the program. (You can change the information after starting your program, but such changes only affect your program the next time you start it.) The information that must be passed to GDB can be categorized into four categories:

arguments.
Specify the arguments to give your program as the arguments of the `run` command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion

or variable substitution) in describing the arguments. On Unix systems, you can control which shell is used with the `SHELL` environment variable. GDB uses the C shell (`/usr/bin/csh`). See ⟨undefined⟩ [Your program's arguments], page ⟨undefined⟩.

environment.
Your program inherits its environment from GDB. However, you can use the GDB commands `set environment` and `unset environment` to change parts of the environment that affect your program. See ⟨undefined⟩ [Your program's environment], page ⟨undefined⟩.

working directory.
Your program inherits its working directory from GDB. You can set the GDB working directory with the `cd` command in GDB. See Section 4.5 [Your program's working directory], page 26.

standard input and output.
Your program as default uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the `run` command line, or you can use the `tty` command to set a different device for your program. See Section 4.6 [Your program's input and output], page 26.

Warning: You can redirect input and output, but you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

Note:

- When you issue the `run` command, your program begins to execute immediately. See Chapter 5 [Stopping and continuing], page 33, for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. See Chapter 8 [Examining Data], page 63.
- If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table, and reads it again. When it does this, GDB tries to retain your current breakpoints.

4.3 Arguments To Your Program

The arguments to your program can be specified by the arguments of the `run` command. On HP-UX, they are passed to the C shell (`/usr/bin/csh`), which expands wildcard characters and performs redirection of I/O, and thence to your program.

On non-Unix systems, the program is usually invoked directly by GDB, which emulates I/O redirection via the appropriate system calls, and the wildcard characters are expanded by the startup code of the program, not by the shell.

The `run` command used with no arguments uses the same arguments used by the previous `run`, or those set by the `set args` command.

Following commands are used to pass the argument values to your program:

`set args` Specify the arguments to be used the next time your program is run. If `set args` has no arguments, `run` executes your program with no arguments. Once you have run your program with arguments, using `set args` before the next `run` is the only way to run it again without arguments.

`show args` Show the arguments to give your program when it is started.

4.4 Program Environment

The *environment* consists of a set of environment variables and their values. Environment variables conventionally record information such as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inherited by all the other programs you run. When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

`show envvar`
: List all the environment variables used by GDB.

`show paths`
: Display the list of search paths for executables (the `PATH` environment variable).

`show environment` [*`varname`*]
: Print the value of environment variable *varname* to be given to your program when it starts. If you do not supply *varname*, print the names and values of all environment variables to be given to your program. You can abbreviate `environment` as `env`.

`set environment` *`varname`* [=*`value`*]
: Set environment variable *varname* to *value*. The value changes for your program only, not for GDB itself. The *value* may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The *value* parameter is optional; if it is eliminated, the variable is set to a null value.

For example, this command:

```
set env USER = foo
```

tells the debugged program, when subsequently run, that its user is named '`foo`'. (The spaces around '`=`' are used for clarity here; they are not actually required.)

`unset environment` *`varname`*
: Remove variable *varname* from the environment to be passed to your program. This is different from '`set env` *`varname`* `=`'; `unset environment` removes the variable from the environment, rather than assigning it an empty value.

`path` *`directory`*
: Add *directory* to the front of the `PATH` environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by whitespace or by a system-dependent separator

character (‘`:`’ on Unix, ‘`;`’ on MS-DOS and MS-Windows). If *directory* is already in the path, it is moved to the front, so it is searched sooner.

You can use the string ‘`$cwd`’ to refer to whatever is the current working directory at the time GDB searches the path. If you use ‘`.`’ instead, it refers to the directory where you executed the `path` command. GDB replaces ‘`.`’ in the *directory* argument (with the current path) before adding *directory* to the search path.

4.5 Working directory

Each time you start your program with `run`, it inherits its working directory from the current working directory of GDB. The GDB working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in GDB with the `cd` command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See Section 12.1 [Commands to specify files], page 103.

Following commands are used to set the working directory for your program:

`cd` *`directory`*
: Set the GDB working directory to *directory*.

`pwd`
: Print the GDB working directory.

4.6 Program Input and Output

By default, the program you run under GDB does input and output to the same terminal that GDB uses. GDB switches the terminal to its own terminal modes to interact with you, but it records the terminal modes your program was using and switches back to them when you continue running your program.

Following commands are used for redirecting the input and output:

`info terminal`
: Displays information recorded by GDB about the terminal modes your program is using.

`tty`
: Another way to specify where your program should do input and output is with the `tty` command. This command accepts a file name as argument, and causes this file to be the default for future `run` commands. It also resets the controlling terminal for the child process, for future `run` commands. For example,

```
tty /dev/ttyb
```

directs that processes started with subsequent `run` commands default to do input and output on the terminal ‘`/dev/ttyb`’ and have that as their controlling terminal.

Note:

- You can redirect your program input and output using shell redirection with the `run` command. For example,

```
run > outfile
```

starts your program, diverting its output to the file '`outfile`'.

- An explicit redirection in `run` overrides the `tty` command's effect on the input/output device, but not its effect on the controlling terminal.
- When you use the `tty` command or redirect input in the `run` command, only the input *for your program* is affected. The input for GDB still comes from your terminal.

4.7 Debugging a Running Process

You can use GDB to debug a running process by specifying the process ID. Following commands are used to debug a running process:

`attach` *`process-id`*

This command attaches to a running process—one that was started outside GDB. (`info files` shows your active targets.) The command takes as argument a process ID. The usual way to find out the process-id of a Unix process is with the `ps` utility, or with the '`jobs -l`' shell command.

`attach` does not repeat if you press ⟨RET⟩ a second time after executing the command.

Note:

- To use `attach`, your program must be running in an environment which supports processes; for example, `attach` does not work for programs on bare-board targets that lack an operating system.
- You must also have permission to send the process a signal.
- When you use `attach`, the debugger finds the program running in the process first by looking in the current working directory, then (if the program is not found) by using the source file search path (see Section 7.3 [Specifying source directories], page 59). You can also use the `file` command to load the program. See Section 12.1 [Commands to Specify Files], page 103.
-

  GDB stops the process being attached for debugging. You can examine and modify an attached process with the GDB commands that are available when you start processes with `run`. You can insert breakpoints; you can step and continue; you can modify storage. See Section 5.1 [Breakpoints in shared libraries], page 33. If you want the process to continue running, you can use the `continue` command after attaching GDB to the process.

`detach`

When you have finished debugging the attached process, you can use the `detach` command to release it from GDB control. The process continues its execution after being detached. After the `detach` command, that process and GDB become completely independent once more, and you are ready to `attach` another process or start one with `run`. `detach` does not repeat if you press ⟨RET⟩ again after executing the command.

If you exit GDB or use the `run` command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see Section 17.6 [Optional warnings and messages], page 252).

> *NOTE:* When GDB attaches to a running program you may get a message saying `"Attaching to process #nnnnn failed."`
>
> The most likely cause for this message is that you have attached to a process that was started across an NFS mount. Versions of the HP-UX kernel before 11.x have a restriction that prevents a debugger from attaching to a process started from an NFS mount, unless the mount was made non-interruptible with the `-nointr` flag, see `mount(1)`.

4.8 Killing the child process

Following command is used to kill the child process:

`kill` Kill the child process in which your program is running under GDB.

The `kill` command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

On some operating systems, a program cannot be executed outside GDB while you have breakpoints set on it inside GDB. You can use the `kill` command in this situation to permit running your program outside the debugger.

The `kill` command is also useful if you wish to recompile and relink your program, since on many systems it is impossible to modify an executable file while it is running in a process. In this case, when you next type `run`, GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current breakpoint settings).

4.9 Debugging programs with multiple threads

In some operating systems, such as HP-UX and Solaris, a single program may have more than one *thread* of execution. The precise semantics of threads differ from one operating system to another, but in general the threads of a single program are akin to multiple processes—except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and private memory.

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- thread-specific breakpoints

> *Warning:* These facilities are not yet available on every GDB configuration where the operating system supports threads. If your GDB does not support threads, these commands have no effect. For example, a system without thread support shows no output from '`info threads`', and always rejects the `thread` command, like this:

```
((gdb)) info threads
((gdb)) thread 1
Thread ID 1 not known.  Use the "info threads" command to
see the IDs of currently known threads.
```

Following commands are used to debug multi-threaded programs:

- '`thread` *`threadno`*', a command to switch among threads
- '`info threads`', a command to inquire about existing threads
- '`thread apply [`*`threadno`*`] [`*`all`*`]` *`args`*', a command to apply a command to a list of threads

The GDB thread debugging facility allows you to observe all threads while your program runs—but whenever GDB takes control, one thread in particular is always the focus of debugging. This thread is called the *current thread*. Debugging commands show program information from the perspective of the current thread.

Whenever GDB detects a new thread in your program, it displays the target system's identification for the thread with a message in the form '`[New` *`systag`*`]`'. *systag* is a thread identifier whose form varies depending on the particular system. For example, on LynxOS, you might see

```
[New process 35 thread 27]
```

when GDB notices a new thread. In contrast, on an SGI system, the *systag* is simply something like '`process 368`', with no further qualifier.

For debugging purposes, GDB associates its own thread number—always a single integer—with each thread in your program.

`info threads`

Display a summary of all threads currently in your program. GDB displays for each thread (in this order):

1. the thread number assigned by GDB
2. the target system's thread identifier (*systag*)
3. the current stack frame summary for that thread

An asterisk '`*`' to the left of the GDB thread number indicates the current thread.

For example,

```
((gdb)) info threads
  3 process 35 thread 27  0x34e5 in sigpause ()
  2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1, argv=0x7ffffff8)
    at threadtest.c:68
```

On HP-UX systems:

For debugging purposes, GDB associates its own thread number—a small integer assigned in thread-creation order—with each thread in your program.

Whenever GDB detects a new thread in your program, it displays both GDB's thread number and the target system's identification for the thread with a message in the form '`[New` *`systag`*`]`'. *systag* is a thread identifier whose form varies depending on the particular system. For example, on HP-UX, you see

```
[New thread 2 (system thread 26594)]
```

when GDB notices a new thread.

On HP-UX systems, you can control the display of thread creation messages. Following commands are used to control the display of thread creation:

`set threadverbose on`
: Enable the output of informational messages regarding thread creation. The default setting is `on`. You can set it to `off` to stop the display of messages.

`set threadverbose off`
: Disable the output of informational messages regarding thread creation. The default setting is `on`. You can set it to `on` to display messages.

`show threadverbose`
: Display whether set threadverbose is `on` or `off`.

Here are commands to get more information about threads:

`info threads`
: Display a summary of all threads currently in your program. GDB displays for each thread (in this order):

1. the thread number assigned by GDB
2. the target system's thread identifier (*systag*)
3. the current stack frame summary for that thread
4. the priority of a thread

An asterisk '`*`' to the left of the GDB thread number indicates the current thread.

For example,

```
((gdb)) info threads
    * 3 system thread 26607  worker (wptr=0x7b09c318 "@") \
                               at quicksort.c:137
      2 system thread 26606  0x7b0030d8 in __ksleep () \
                               from /usr/lib/libc.2
      1 system thread 27905  0x7b003498 in _brk () \
                               from /usr/lib/libc.2
```

`thread *threadno*`
: Make thread number *threadno* the current thread. The command argument *threadno* is the internal GDB thread number, as shown in the first field of the '`info threads`' display. GDB responds by displaying the system identifier of the thread you selected, and its current stack frame summary:

```
((gdb)) thread 2
[Switching to thread 2 (system thread 26594)]
0x34e5 in sigpause ()
```

As with the '`[New ...]`' message, the form of the text after '`Switching to`' depends on your system's conventions for identifying threads.

`thread apply [`*`threadno`*`] [`*`all`*`]` *`args`*

The `thread apply` command allows you to apply a command to one or more threads. Specify the numbers of the threads that you want affected with the command argument *threadno*. *threadno* is the internal GDB thread number, as shown in the first field of the '`info threads`' display. To apply a command to all threads, use `thread apply all` *args*.

Whenever GDB stops your program, due to a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. GDB alerts you to the context switch with a message of the form '`[Switching to` *`systag`*`]`' to identify the thread.

See Section 5.4 [Stopping and starting multi-thread programs], page 48, for more information about how GDB behaves when you stop and start programs with multiple threads.

See ⟨undefined⟩ [Setting watchpoints], page ⟨undefined⟩, for information about watchpoints in programs with multiple threads.

Note:

On HP-UX 11.x, debugging a multi-thread process can cause a deadlock if the process is waiting for an NFS-server response. A thread can be stopped while asleep in this state, and NFS holds a lock on the rnode while asleep.

To prevent the thread from being interrupted while holding the rnode lock, make the NFS mount non-interruptible with the '`-nointr`' flag. See mount(1).

4.10 Debugging programs with multiple processes

On most systems, GDB has no special support for debugging programs which create additional processes using the `fork` function. When a program forks, GDB will continue to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a `SIGTRAP` signal which (unless it catches the signal) will cause it to terminate.

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call to `sleep` in the code which the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the `ps` program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see Section 4.7 [Attach], page 27). From that point on you can debug the child process just like any other process which you attached to.

On HP-UX (11.x and later only), GDB provides support for debugging programs that create additional processes using the `fork` or `vfork` function.

By default, when a program forks, GDB will continue to debug the parent process and the child process will run unimpeded.

If you want to follow the child process instead of the parent process, use the command `set follow-fork-mode`.

`set follow-fork-mode` *`mode`*

Set the debugger response to a program call of `fork` or `vfork`. A call to `fork` or `vfork` creates a new process. The *mode* can be:

`parent` The original process is debugged after a fork. The child process runs unimpeded. This is the default.

`child` The new process is debugged after a fork. The parent process runs unimpeded.

`show follow-fork-mode`
Display the current debugger response to a `fork` or `vfork` call.

If you ask to debug a child process and a `vfork` is followed by an `exec`, GDB executes the new target up to the first breakpoint in the new target. If you have a breakpoint set on `main` in your original program, the breakpoint will also be set on the child process's `main`.

When a child process is spawned by `vfork`, you cannot debug the child or parent until an `exec` call completes.

If you issue a `run` command to GDB after an `exec` call executes, the new target restarts. To restart the parent process, use the `file` command with the parent executable name as its argument.

You can use the `catch` command to make GDB stop whenever a `fork`, `vfork`, or `exec` call is made. See Section 5.1.2 [Setting catchpoints], page 37.

5 Stopping and Continuing

The principal purpose of a debugger is to let you stop your program before it terminates abnormaly or runs into trouble, so that you can investigate and determine the reason.

Inside GDB, your program can stop for several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You can then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide information on the status of your program—but you can also explicitly request this information at any time.

`info program`
: Display information about the status of your program: whether it is running or not, what process it is, and why it stopped.

5.1 Breakpoints

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break` command and its variants. (see Section 5.1.1 [Setting breakpoints], page 33) You can stop your program by line number, function name or an address in the program.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See Section 8.6 [Automatic display], page 68.

In HP-UX, SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can set breakpoints in shared libraries before the executable is run. See Section 14.20 [Stopping and starting in shared libraries], page 174.

A *catchpoint* is another special breakpoint that stops your program when a certain kind of event occurs, such as the throwing of a C++ exception or the loading of a library. As with watchpoints, you use a different command to set a catchpoint (see Section 5.1.2 [Setting catchpoints], page 37), but aside from that, you can manage a catchpoint like any other breakpoint. (To stop when your program receives a signal, use the `handle` command; see Section 5.3 [Signals], page 46.)

GDB assigns a number to each breakpoint, watchpoint, or catchpoint when you create it; these numbers are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on your program until you enable it again.

Some GDB commands accept a range of breakpoints on which to operate. A breakpoint range is either a single breakpoint number, like '`5`', or two such numbers, in increasing order, separated by a hyphen, like '`5-7`'. When a breakpoint range is given to a command, all breakpoint in that range are operated on.

5.1.1 Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable '`$bpnum`' records the number of the breakpoint you've set most recently;

see Section 8.9 [Convenience variables], page 75, for a discussion of what you can do with convenience variables.

You have several ways to say where the breakpoint should go.

`break` *`function`*
: Set a breakpoint at entry to function *function*. When using source languages that permit overloading of symbols, such as C++, *function* may refer to more than one possible place to break. See Section 5.1.7 [Breakpoint menus], page 42, for a discussion of that situation.

`break +`*`offset`*
`break -`*`offset`*
: Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected *stack frame*. (See Section 6.1 [Frames], page 51, for a description of stack frames.)

`break` *`linenum`*
: Set a breakpoint at line *linenum* in the current source file. The current source file is the last file whose source text was printed. The breakpoint will stop your program just before it executes any of the code on that line.

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

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

`break *`*`address`*
: Set a breakpoint at address *address*. You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

`break`
: When called without any arguments, `break` sets a breakpoint at the next instruction to be executed in the selected stack frame (see Chapter 6 [Examining the Stack], page 51). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a `finish` command in the frame inside the selected frame—except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops.

  GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when your program stopped.

`break ... if` *`cond`*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. '`...`' stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.1.5 [Break conditions], page 40, for more information on breakpoint conditions.

`tbreak` *args*
: Set a breakpoint enabled only for one stop. *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See Section 5.1.4 [Disabling breakpoints], page 39.

`hbreak` *args*
: Set a hardware-assisted breakpoint. *args* are the same as for the `break` command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClite DSU and some x86-based targets. These targets will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can take a limited number of breakpoints. For example, on the DSU, only two data breakpoints can be set at a time, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones (see Section 5.1.4 [Disabling], page 39). See Section 5.1.5 [Break conditions], page 40.

`thbreak` *args*
: Set a hardware-assisted breakpoint enabled only for one stop. *args* are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support and some target hardware may not have this support. See Section 5.1.4 [Disabling breakpoints], page 39. See also Section 5.1.5 [Break conditions], page 40.

`rbreak` *regex*
: Set breakpoints on all functions matching the regular expression *regex*. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the `break` command. You can delete them, disable them, or make them conditional the same way as any other breakpoint.

  The syntax of the regular expression is the standard one used with tools like '`grep`'. Note that this is different from the syntax used by shells, so for instance `foo*` matches all functions that include an `fo` followed by zero or more `o`s. There is an implicit `.*` leading and trailing the regular expression you supply, so to match only functions that begin with `foo`, use `^foo`.

  When debugging C++ programs, `rbreak` is useful for setting breakpoints on overloaded functions that are not members of any special classes.

`info breakpoints` [*n*]
`info break` [*n*]
`info watchpoints` [*n*]
: Print a table of all breakpoints, watchpoints, and catchpoints set and not deleted, with the following columns for each breakpoint:

Breakpoint Numbers

Type Breakpoint, watchpoint, or catchpoint.

Disposition
: Whether the breakpoint is marked to be disabled or deleted when hit.

Enabled or Disabled
: Enabled breakpoints are marked with '`y`'. '`n`' marks breakpoints that are not enabled.

Address Where the breakpoint is in your program, as a memory address.

What Where the breakpoint is in the source for your program, as a file and line number.

If a breakpoint is conditional, `info break` shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that.

`info break` with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see Section 8.5 [Examining memory], page 67).

`info break` displays a count of the number of times the breakpoint has been hit. This is especially useful in conjunction with the `ignore` command. You can ignore a large number of breakpoint hits, look at the breakpoint info to see how many times the breakpoint was hit, and then run again, ignoring one less than that number. This will get you quickly to the last hit of that breakpoint.

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see Section 5.1.5 [Break conditions], page 40).

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of `longjmp` (in C programs). These internal breakpoints are assigned negative numbers, starting with `-1`; '`info breakpoints`' does not display them.

You can see these breakpoints with the GDB maintenance command '`maint info breakpoints`'.

`maint info breakpoints`
: Using the same format as '`info breakpoints`', display both the breakpoints you've set explicitly, and those GDB is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

  `breakpoint`
  : Normal, explicitly set breakpoint.

  `watchpoint`
  : Normal, explicitly set watchpoint.

  `longjmp` Internal breakpoint, used to handle correctly stepping through `longjmp` calls.

`longjmp resume`
: Internal breakpoint at the target of a `longjmp`.

`until`
: Temporary internal breakpoint used by the GDB `until` command.

`finish`
: Temporary internal breakpoint used by the GDB `finish` command.

`shlib events`
: Shared library events.

5.1.2 Setting catchpoints

You can use *catchpoints* to cause the debugger to stop for certain kinds of program events, such as C++ exceptions or the loading of a shared library. Use the `catch` command to set a catchpoint.

`catch` *`event`*

Stop when *event* occurs. *event* can be any of the following:

`throw`
: The throwing of a C++ exception.

`catch`
: The catching of a C++ exception.

`exec`
: A call to `exec`. This is currently only available for HP-UX.

`fork`
: A call to `fork`. This is currently only available for HP-UX.

`vfork`
: A call to `vfork`. This is currently only available for HP-UX.

`load`
`load` *`libname`*
: The dynamic loading of any shared library, or the loading of the library *libname*. This is currently only available for HP-UX.

`unload`
`unload` *`libname`*
: The unloading of any dynamically loaded shared library, or the unloading of the library *libname*. This is currently only available for HP-UX.

`tcatch` *`event`*

Set a catchpoint that is enabled only for one stop. The catchpoint is automatically deleted after the first time the event is caught.

Use the `info break` command to list the current catchpoints.

There are currently some limitations to C++ exception handling (`catch throw` and `catch catch`) in GDB:

- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program either to abort or to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits. This is the case even if you set a catchpoint for the exception; catchpoints on exceptions are disabled within interactive calls.

- You cannot raise an exception interactively.
- You cannot install an exception handler interactively.

Sometimes `catch` is not the best way to debug exception handling: if you need to know exactly where an exception is raised, it is better to stop *before* the exception handler is called, since that way you can see the stack before any unwinding takes place. If you set a breakpoint in an exception handler instead, it may not be easy to find out where the exception was raised.

To stop just before an exception handler is called, you need some knowledge of the implementation. In the case of GNU C++, exceptions are raised by calling a library function named `__raise_exception` which has the following ANSI C interface:

```
/* addr is where the exception identifier is stored.
   id is the exception identifier.  */
void __raise_exception (void **addr, void *id);
```

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__raise_exception` (see Section 5.1 [Breakpoints; watchpoints; and exceptions], page 33).

With a conditional breakpoint (see Section 5.1.5 [Break conditions], page 40) that depends on the value of *id*, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

5.1.3 Deleting breakpoints

It is often necessary to eliminate a breakpoint, watchpoint, or catchpoint once it has done its job and you no longer want your program to stop there. This is called *deleting* the breakpoint. A breakpoint that has been deleted no longer exists; it is forgotten.

With the `clear` command you can delete breakpoints according to where they are in your program. With the `delete` command you can delete individual breakpoints, watchpoints, or catchpoints by specifying their breakpoint numbers.

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

`clear`
: Delete any breakpoints at the next instruction to be executed in the selected stack frame (see Section 6.5 [Selecting a frame], page 53). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

`clear` *`function`*
`clear` *`filename`*`:`*`function`*
: Delete any breakpoints set at entry to the function *function*.

`clear` *`linenum`*
`clear` *`filename`*`:`*`linenum`*
: Delete any breakpoints set at or within the code of the specified line.

`delete` [`breakpoints`] [*range*...]
: Delete the breakpoints, watchpoints, or catchpoints of the breakpoint ranges specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have `set confirm off`). You can abbreviate this command as `d`.

5.1.4 Disabling breakpoints

Rather than deleting a breakpoint, watchpoint, or catchpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later.

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

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

- Enabled. The breakpoint stops your program. A breakpoint set with the `break` command starts out in this state.
- Disabled. The breakpoint has no effect on your program.
- Enabled once. The breakpoint stops your program, but then becomes disabled.
- Enabled for deletion. The breakpoint stops your program, but immediately after it does so it is deleted permanently. A breakpoint set with the `tbreak` command starts out in this state.

You can use the following commands to enable or disable breakpoints, watchpoints, and catchpoints:

`disable` [`breakpoints`] [*range*...]
: Disable the specified breakpoints—or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate `disable` as `dis`.

`enable` [`breakpoints`] [*range*...]
: Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

`enable` [`breakpoints`] `once` *range*...
: Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately after stopping your program.

`enable` [`breakpoints`] `delete` *range*...
: Enable the specified breakpoints to work once, then die. GDB deletes any of these breakpoints as soon as your program stops there.

Except for a breakpoint set with `tbreak` (see Section 5.1.1 [Setting breakpoints], page 33), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see Section 5.2 [Continuing and stepping], page 44.)

5.1.5 Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language (see Section 8.1 [Expressions], page 63). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated—that is, when the condition is false. In C, if you want to test an assertion expressed by the condition *assert*, you should set the condition '`! assert`' on the appropriate breakpoint.

Conditions are also accepted for watchpoints; you may not need them, since a watchpoint is inspecting the value of an expression anyhow—but it might be simpler, say, to just set a watchpoint on a variable name, and specify a condition that tests whether the new value is an interesting one.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached (see Section 5.1.6 [Breakpoint command lists], page 41).

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the `break` command. See Section 5.1.1 [Setting breakpoints], page 33. They can also be changed at any time with the `condition` command.

You can also use the `if` keyword with the `watch` command. The `catch` command does not recognize the `if` keyword; `condition` is the only way to impose a further condition on a catchpoint.

`condition` *`bnum expression`*

Specify *expression* as the break condition for breakpoint, watchpoint, or catchpoint number *bnum*. After you set a condition, breakpoint *bnum* stops your program only if the value of *expression* is true (nonzero, in C). When you use `condition`, GDB checks *expression* immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. If *expression* uses symbols not referenced in the context of the breakpoint, GDB prints an error message:

```
No symbol "foo" in current context.
```

GDB does not actually evaluate *expression* at the time the `condition` command (or a command that sets a breakpoint with a condition, like `break if ...`) is given, however. See Section 8.1 [Expressions], page 63.

`condition` *bnum*
: Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

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

`ignore` *bnum count*
: Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, your program's execution does not stop; other than to decrement the ignore count, GDB takes no action.

  To make the breakpoint stop the next time it is reached, specify a count of zero.

  When you use `continue` to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to `continue`, rather than using `ignore`. See Section 5.2 [Continuing and stepping], page 44.

  If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, GDB resumes checking the condition.

  You could achieve the effect of the ignore count with a condition such as '`$foo-- <= 0`' using a debugger convenience variable that is decremented each time. See Section 8.9 [Convenience variables], page 75.

Ignore counts apply to breakpoints, watchpoints, and catchpoints.

5.1.6 Breakpoint command lists

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

`commands` [*bnum*]
`...` *command-list* `...`
`end`
: Specify a list of commands for breakpoint number *bnum*. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands.

  To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands.

  With no *bnum* argument, `commands` refers to the last breakpoint, watchpoint, or catchpoint set (not to the breakpoint most recently encountered).

Pressing ⟨RET⟩ as a means of repeating the last GDB command is disabled within a *command-list*.

You can use breakpoint commands to start your program up again. Simply use the `continue` command, or `step`, or any other command that resumes execution.

Any other commands in the command list, after a command that resumes execution, are ignored. This is because any time you resume execution (even with a simple `next` or `step`), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If the first command you specify in a command list is `silent`, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the remaining commands print anything, you see no sign that the breakpoint was reached. `silent` is meaningful only at the beginning of a breakpoint command list.

The commands `echo`, `output`, and `printf` allow you to print precisely controlled output, and are often useful in silent breakpoints. See Section 18.4 [Commands for controlled output], page 258.

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

```
break foo if x>0
commands
silent
printf "x is %d\n",x
cont
end
```

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

```
break 403
commands
silent
set x = y + 4
cont
end
```

5.1.7 Breakpoint menus

Some programming languages (notably C++) permit a single function name to be defined several times, for application in different contexts. This is called *overloading*. When a function name is overloaded, '`break` *`function`*' is not enough to tell GDB where you want a breakpoint. If you realize this is a problem, you can use something like '`break` *`function`*`(`*`types`*`)`' to specify which particular version of the function you want. Otherwise, GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt '`>`'. The first two options are always '`[0] cancel`' and

'`[1] all`'. Typing *1* sets a breakpoint at each definition of *function*, and typing *0* aborts the `break` command without setting any new breakpoints.

For example, the following session excerpt shows an attempt to set a breakpoint at the overloaded symbol `String::after`. We choose three particular definitions of that function name:

```
((gdb)) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
 breakpoints.
((gdb))
```

5.1.8 "Cannot insert breakpoints"

Under some operating systems, breakpoints cannot be used in a program if any other process is running that program. In this situation, attempting to run or continue a program with a breakpoint causes GDB to print an error message:

```
Cannot insert breakpoints.
The same program may be running in another process.
```

When this happens, you have three ways to proceed:

1. Remove or disable the breakpoints, then continue.
2. Suspend GDB, and copy the file containing your program to a new name. Resume GDB and use the `exec-file` command to specify that GDB should run your program under that name. Then start your program again.
3. Relink your program so that the text segment is nonsharable, using the linker option '`-N`'. The operating system limitation may not apply to nonsharable executables.

A similar message can be printed if you request too many active hardware-assisted breakpoints and watchpoints:

```
Stopped; cannot insert breakpoints.
You may have requested too many hardware breakpoints and watchpoints.
```

This message is printed when you attempt to resume the program, since only then GDB knows exactly how many hardware breakpoints and watchpoints it needs to insert.

When this message is printed, you need to disable or remove some of the hardware-assisted breakpoints and watchpoints, and then continue.

5.2 Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If it stops due to a signal, you may want to use `handle`, or use '`signal 0`' to resume execution. See Section 5.3 [Signals], page 46.)

`continue` [*ignore-count*]
`c` [*ignore-count*]
`fg` [*ignore-count*]

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore` (see Section 5.1.5 [Break conditions], page 40).

The argument *ignore-count* is meaningful only when your program stopped due to a breakpoint. At other times, the argument to `continue` is ignored.

The synonyms `c` and `fg` (for *foreground*, as the debugged program is deemed to be the foreground program) are provided purely for convenience, and have exactly the same behavior as `continue`.

To resume execution at a different place, you can use `return` (see Section 11.4 [Returning from a function], page 99) to go back to the calling function; or `jump` (see Section 11.2 [Continuing at a different address], page 98) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see Section 5.1 [Breakpoints; watchpoints; and catchpoints], page 33) at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

`step`

Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated `s`.

> *Warning:* If you use the `step` command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it will not step into a function which is compiled without debugging information. To step through functions without debugging information, use the `stepi` command, described below.

The `step` command only stops at the first instruction of a source line. This prevents the multiple stops that could otherwise occur in switch statements, for loops, etc. `step` continues to stop if a function that has debugging information is called within the line. In other words, `step` *steps inside* any functions called within the line.

Also, the `step` command only enters a function if there is line number information for the function. Otherwise it acts like the `next` command. This avoids problems when using `cc -gl` on MIPS machines. Previously, `step` entered subroutines if there was any debugging information about the routine.

`step` *count*
: Continue running as in `step`, but do so *count* times. If a breakpoint is reached, or a signal not related to stepping occurs before *count* steps, stepping stops right away.

`next` [*count*]
: Continue to the next source line in the current (innermost) stack frame. This is similar to `step`, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next` command. This command is abbreviated `n`.

An argument *count* is a repeat count, as for `step`.

The `next` command only stops at the first instruction of a source line. This prevents multiple stops that could otherwise occur in switch statements, for loops, etc.

`finish`
: Continue running until just after function in the selected stack frame returns. Print the returned value (if any).

Contrast this with the `return` command (see Section 11.4 [Returning from a function], page 99).

`until`
`u`
: Continue running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once. It is like the `next` command, except that when `until` encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

This means that when you reach the end of a loop after single stepping though it, `until` makes your program continue execution until it exits the loop. In contrast, a `next` command at the end of a loop simply steps back to the beginning of the loop, which forces you to step through the next iteration.

`until` always stops your program if it attempts to exit the current stack frame.

`until` may produce somewhat counterintuitive results if the order of machine code does not match the order of the source lines. For example, in the following excerpt from a debugging session, the `f` (`frame`) command shows that execution is stopped at line `206`; yet when we use `until`, we get to line `195`:

```
((gdb)) f
#0  main (argc=4, argv=0xf7fffae8) at m4.c:206
206                 expand_input();
((gdb)) until
195             for ( ; argc > 0; NEXTARG) {
```

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop—

even though the test in a C `for`-loop is written before the body of the loop. The `until` command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement—not in terms of the actual machine code.

`until` with no argument works by means of single instruction stepping, and hence is slower than `until` with an argument.

`until` *`location`*
`u` *`location`*

Continue running your program until either the specified location is reached, or the current stack frame returns. *location* is any of the forms of argument acceptable to `break` (see Section 5.1.1 [Setting breakpoints], page 33). This form of the command uses breakpoints, and hence is quicker than `until` without an argument.

`stepi`
`stepi` *`arg`*
`si`

Execute one machine instruction, then stop and return to the debugger.

It is often useful to do '`display/i $pc`' when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See Section 8.6 [Automatic display], page 68.

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

`nexti`
`nexti` *`arg`*
`ni`

Execute one machine instruction, but if it is a function call, proceed until the function returns.

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

5.3 Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in Unix `SIGINT` is the signal a program gets when you type an interrupt character (often *`C-c`*); `SIGSEGV` is the signal a program gets from referencing a place in memory far away from all the areas in use; `SIGALRM` occurs when the alarm clock timer goes off (which happens only if your program has requested an alarm).

Some signals, including `SIGALRM`, are a normal part of the functioning of your program. Others, such as `SIGSEGV`, indicate errors; these signals are *fatal* (they kill your program immediately) if the program has not specified in advance some other way to handle the signal. `SIGINT` does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like `SIGALRM` (so as not to interfere with their role in the functioning of your program) but to stop your program immediately whenever an error signal happens. You can change these settings with the `handle` command.

> *Note:* Use caution if you disable all signals from certain processes. Disabling '`SIGTRAP`' in your program may cause your program to hang.
>
> HP-UX uses '`SIGTRAP`' to communicate with the debugger. If you disable all signals from certain processes so that signals will be delivered to the right process, your program may hang when you try to debug it. This behavior occurs because if you disable '`SIGTRAP`', the debugger no longer receives notification of events such as breakpoint hits and loading or unloading of shared libraries.
>
> To prevent this problem:
>
> Make certain you set this flag:
>
> ```
> (gdb) set complain-if-sigtrap-disabled on
> ```
>
> Also make certain the following warning was not emitted by the debugger before your program hung:
>
> ```
> Warning: Thread %d (in process %d) has disabled SIGTRAPs.
> Debugging this thread is probably impossible.
> If you do not want to see this message again, use:
> "set complain-if-sigtrap-disabled 0"
> ```

`info signals`
`info handle`
: Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

  `info handle` is an alias for `info signals`.

`handle` *`signal keywords...`*
: Change the way GDB handles signal *signal*. *signal* can be the number of a signal or its name (with or without the '`SIG`' at the beginning). The *keywords* say what change to make.

The keywords allowed by the `handle` command can be abbreviated. Their full names are:

`nostop` GDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.

`stop` GDB should stop your program when this signal happens. This implies the `print` keyword as well.

`print` GDB should print a message when this signal happens.

`noprint` GDB should not mention the occurrence of the signal at all. This implies the `nostop` keyword as well.

`pass` GDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled.

`nopass` GDB should not allow your program to see this signal.

When a signal stops your program, the signal is not visible to the program until you continue. Your program sees the signal then, if `pass` is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can use the `handle` command with `pass` or `nopass` to control whether your program sees that signal when you continue.

You can also use the `signal` command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with '`signal 0`'. See Section 11.3 [Giving your program a signal], page 99.

5.4 Stopping and starting multi-thread programs

When your program has multiple threads (see Section 4.9 [Debugging programs with multiple threads], page 28), you can choose whether to set breakpoints on all threads, or on a particular thread.

`break` *`linespec`* `thread` *`threadno`*
`break` *`linespec`* `thread` *`threadno`* `if ...`

> *linespec* specifies source lines; there are several ways of writing them, but the effect is always to specify some source line.
>
> Use the qualifier '`thread` *`threadno`*' with a breakpoint command to specify that you only want GDB to stop the program when a particular thread reaches this breakpoint. *threadno* is one of the numeric thread identifiers assigned by GDB, shown in the first column of the '`info threads`' display.
>
> If you do not specify '`thread` *`threadno`*' when you set a breakpoint, the breakpoint applies to *all* threads of your program.
>
> You can use the `thread` qualifier on conditional breakpoints as well; in this case, place '`thread` *`threadno`*' before the breakpoint condition, like this:
>
> ```
> ((gdb)) break frik.c:13 thread 28 if bartab > lim
> ```

Whenever your program stops under GDB for any reason, *all* threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, including switching between threads, without worrying that things may change underfoot.

Conversely, whenever you restart the program, *all* threads start executing. *This is true even when single-stepping* with commands like `step` or `next`.

In particular, GDB cannot single-step all threads in lockstep. Since thread scheduling is up to your debugging target's operating system (not controlled by GDB), other threads may execute more than one statement while the current thread completes a single step. Moreover, in general other threads stop in the middle of a statement, rather than at a clean statement boundary, when the program stops.

You might even find your program stopped in another thread after continuing or even single-stepping. This happens whenever some other thread runs into a breakpoint, a signal, or an exception before the first thread completes whatever you requested.

On some OSes, you can lock the OS scheduler and thus allow only a single thread to run.

`set scheduler-locking` *`mode`*
: Set the scheduler locking mode. If it is `off`, then there is no locking and any thread may run at any time. If `on`, then only the current thread may run when the inferior is resumed. The `step` mode optimizes for single-stepping. It stops other threads from "seizing the prompt" by preempting the current thread while you are stepping. Other threads will only rarely (or never) get a chance to run when you step. They are more likely to run when you '`next`' over a function call, and they are completely free to run when you use commands like '`continue`', '`until`', or '`finish`'. However, unless another thread hits a breakpoint during its timeslice, they will never steal the GDB prompt away from the thread that you are debugging.

`show scheduler-locking`
: Display the current scheduler locking mode.

6 Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated.The information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*.

The GDB commands for examining the stack allow you to view all of this information.

6.1 Stack frames

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

When your program is started, the stack has only one frame, that of the function `main`. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

One of the stack frames is *selected* by GDB and the GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in. See Section 6.5 [Selecting a frame], page 53.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command (see Section 6.6 [Information about a frame], page 54).

6.2 Stacks Without frames

Some compilers provide a way to compile functions so that they operate without stack frames. (For example, the gcc option

`‘-fomit-frame-pointer’`

generates functions without a frame.) This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations. If the innermost function invocation has no stack frame, GDB nevertheless regards it as though it had a separate frame, which is numbered zero as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

6.3 Commands for Examining the Stack

The following commands are used for examining the stack:

`frame` *`args`*
: Select and print a stack frame. With no argument, prints the selected stack frame. An argument specifies the frame to select. It can be a stack frame number or the address of the frame. With argument, nothing is printed if input is coming from a command file or a user-defined command.

`select-frame`
: The `select-frame` command allows you to move from one stack frame to another without printing the frame. This is the silent version of `frame`.

6.4 Backtraces

A backtrace is a report of the active stack frames instantiated by the execution of a program. It shows one line per frame, for all the active frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

The following commands are used for backtrace:

`backtrace`
`bt`
: Print a backtrace of the entire stack: one line per frame for all frames in the stack.

  You can stop the backtrace at any time by typing the system interrupt character, normally *`C-c`*.

`backtrace` *`n`*
`bt` *`n`*
: Similar, but print only the innermost *n* frames.

`backtrace -`*`n`*
`bt -`*`n`*
: Similar, but print only the outermost *n* frames.

`backtrace-other-thread`
: Print backtrace of all stack frames for a thread with stack pointer SP and program counter PC. This command is useful in cases where the debugger does not support a user thread package fully.

The names `where` and `info stack` (abbreviated `info s`) are additional aliases for `backtrace`.

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use `set print address off`. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

Here is an example of a backtrace. It was made with the command '`bt 3`', so it shows the innermost three frames.

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
(More stack frames follow...)
```

The display for frame zero does not begin with a program counter value, indicating that your program has stopped at the beginning of the code for line `993` of `builtin.c`.

6.5 Selecting a frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment.

The following commands are used for selecting a stack frame; all of them finish by printing a brief description of the stack frame selected.

`frame` *n*
`f` *n*
: Select frame number *n*. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is the one for `main`.

`frame` *addr*
`f` *addr*
: Select the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful when your program has multiple stacks and switches between them.

  Note:

  - On the SPARC architecture, `frame` needs two addresses to select an arbitrary frame: a frame pointer and a stack pointer.
  - On the MIPS and Alpha architecture, it needs two addresses: a stack pointer and a program counter.
  - On the 29k architecture, it needs three addresses: a register stack pointer, a program counter, and a memory stack pointer.

`up` *n*
: Move *n* frames up the stack. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one.

`down` *n*
: Move *n* frames down the stack. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one. You may abbreviate `down` as `do`.

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the source file and line number of execution in that frame. The second line shows the text of that source line.

For example:

```
((gdb)) up
#1  0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc)
    at env.c:10
10              read_input_file (argv[i]);
```

After such a printout, the `list` command with no arguments prints ten lines centered on the point of execution in the frame. See Section 7.1 [Printing source lines], page 57.

`up-silently` *n*
`down-silently` *n*
: These two commands are variants of `up` and `down`, respectively; they differ in that they do their work silently, without causing display of the new frame. They are intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting.

6.6 Information about a frame

The following commands are used to print information about the selected stack frame:

`frame`
`f`
: When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated `f`. With an argument, this command is used to select a stack frame. See Section 6.5 [Selecting a frame], page 53.

`info frame`
`info f`
: This command prints a verbose description of the selected stack frame, including:

  - the address of the frame
  - the address of the next frame down (called by this frame)
  - the address of the next frame up (caller of this frame)
  - the language in which the source code corresponding to this frame is written
  - the address of the frame's arguments
  - the address of the frame's local variables
  - the program counter saved in it (the address of execution in the caller frame)
  - which registers were saved in the frame

  The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

`info frame` *addr*
`info f` *addr*
: Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command. This requires

the same kind of address (more than one for some architectures) that you specify in the `frame` command. See Section 6.5 [Selecting a frame], page 53.

`info args` Print the arguments of the selected frame, each on a separate line.

`info locals`
: Print the local variables of the selected frame, each on a separate line. These are all variables (declared either static or automatic) accessible at the point of execution of the selected frame.

`info catch`
: Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the `up`, `down`, or `frame` commands); then type `info catch`. See Section 5.1.2 [Setting catchpoints], page 37.

7 Examining Source Files

GDB can print parts of the source code of your program, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see Section 6.5 [Selecting a frame], page 53), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

You can invoke GDB from its GNU Emacs interface to view the source code see Chapter 19 [Using GDB under GNU Emacs], page 259.

7.1 Printing source lines

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

The following forms of the `list` command are used:

`list` *`linenum`*
: Prints lines centered around line number *linenum* in the current source file.

`list` *`function`*
: Prints lines centered around the beginning of function *function*.

`list`
: Prints more lines. The `list` command prints lines following the lines printed by a previously executed `list` command. If the command prior to executing a `list` just printed the stack frame, then the `list` command only prints the lines around that line.

`list -`
: Prints lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the `list` command.

The number of lines printed by GDB can be set by the `set listsize` command. The following two forms are supported:

`set listsize` *`count`*
: Makes the `list` command display *count* source lines (unless the `list` argument explicitly specifies some other number).

`show listsize`
: Displays the number of lines that `list` prints.

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

In general, the `list` command expects you to supply zero, one or two *linespecs*.

Linespecs specify source lines. There are several ways of writing them, but the most common way is to specify some source line.

The following arguments can be given to the `list` command:

`list` *`linespec`*
: Print lines centered around the line specified by *linespec*.

`list` *`first`*`,`*`last`*
: Print lines from *first* to *last*. Both arguments must be linespecs.

`list ,`*`last`*
: Print lines ending with *last*.

`list` *`first`*`,`
: Print lines starting with *first*.

`list +` Print lines just after the lines last printed.

`list -` Print lines just before the lines last printed.

`list` As described in the preceding table.

A single source line can be specified in the following ways:

`number` Specifies line *number* of the current source file. When a `list` command has two linespecs, this refers to the same source file as the first linespec.

`+`*`offset`* Specifies the line *offset* lines after the last line printed. When used as the second linespec in a `list` command that has two, this specifies the line *offset* lines down from the first linespec.

`-`*`offset`* Specifies the line *offset* lines before the last line printed.

`filename``:`*`number`*
: Specifies line *number* in the source file *filename*.

`function` Specifies the line that begins the body of the function *function*. For example: in C, this is the line with the open brace.

`filename``:`*`function`*
: Specifies the line of the open-brace that begins the body of the function *function* in the file *filename*. You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.

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

7.2 Searching source files

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

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

`reverse-search` *regexp*

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

7.3 Specifying source directories

Executable programs sometimes do not record the directories of the source files from which they were compiled. Even when they do, the directories can be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB looks for a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name. Note that the executable search path is *not* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

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

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

When you start GDB, its source path includes only '`cdir`' and '`cwd`', in that order.

To add other directories, you can use the `directory` command.

`directory` *dirname* `...`
`dir` *dirname* `...`

Add directory *dirname* to the front of the source path. Several directory names may be given to this command, separated by '`:`' ('`;`' on MS-DOS and MS-Windows, where '`:`' usually appears as part of absolute file names) or a whitespace. You can specify a directory that is already in the source path; this moves it forward, so GDB searches it sooner.

You can use the string '`$cdir`' to refer to the compilation directory (if one is recorded), and '`$cwd`' to refer to the current working directory. '`$cwd`' is not the same as '`.`'. The former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

`directory`

Reset the source path to empty again. This requires confirmation from the user.

`show directories`

Print the source path and display the directories it contains.

If your source path is cluttered with directories that are no longer of interest, GDB can end up detecting the wrong version of the source. To correct this situation, follow these steps:

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

7.4 Source and machine code

You can use the command `info line` to map source lines to program addresses (and vice versa), and the command `disassemble` to display a range of addresses as machine instructions. When run under GNU Emacs mode, the `info line` command causes the arrow to point to the line specified. Also, `info line` prints addresses in symbolic form as well as hex.

`info line` *`linespec`*
: Print the starting and ending addresses of the compiled code for source line *linespec*. You can specify source lines in any of the ways understood by the `list` command (see Section 7.1 [Printing source lines], page 57).

For example, we can use `info line` to discover the location of the object code for the first line of function.

`m4_changequote`:

```
((gdb)) info line m4_changequote

Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can also inquire (using `*`*`addr`* as the form for *linespec*) what source line covers a particular address. For example,

```
((gdb)) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

After `info line`, the default address for the `x` command is changed to the starting address of the line, so that '`x/i`' is sufficient to begin examining the machine code (see Section 8.5 [Examining memory], page 67). Also, this address is saved as the value of the convenience variable `$_` (see Section 8.9 [Convenience variables], page 75).

`disassemble`
: This specialized command dumps a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; GDB dumps the function surrounding this value. Two arguments specify a range of addresses (first inclusive, second exclusive) to dump.

The following example shows the disassembly of a range of addresses of HP PA-RISC 2.0 code:

```
((gdb)) disas 0x32c4 0x32e4
Dump of assembler code from 0x32c4 to 0x32e4:
0x32c4 <main+204>:      addil 0,dp
0x32c8 <main+208>:      ldw 0x22c(sr0,r1),r26
0x32cc <main+212>:      ldil 0x3000,r31
0x32d0 <main+216>:      ble 0x3f8(sr4,r31)
0x32d4 <main+220>:      ldo 0(r31),rp
0x32d8 <main+224>:      addil -0x800,dp
0x32dc <main+228>:      ldo 0x588(r1),r26
0x32e0 <main+232>:      ldil 0x3000,r31
End of assembler dump.
```

Some architectures have more than one commonly-used set of instruction mnemonics or other syntax.

`set disassembly-flavor` *`instruction-set`*

: Select the instruction set to use when disassembling the program via the `disassemble` or `x/i` commands.

  Currently this command is only defined for the Intel x86 family. You can set *instruction-set* to either `intel` or `att`. The default is `att`, the AT&T flavor used by default by Unix assemblers for x86-based targets.

8 Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym `inspect`. It evaluates and prints the value of an expression of the language your program is written in (see Chapter 9 [Using GDB with Different Languages], page 79).

The following forms of `print` command are supported:

`print` *`expr`*
`print /`*`f expr`*
: *expr* is an expression (in the source language). By default the value of *expr* is printed in a format appropriate to its data type; you can choose a different format by specifying '`/`*`f`*', where *f* is a letter specifying the format; see Section 8.4 [Output formats], page 66.

`print`
`print /`*`f`*
: If you omit *expr*, GDB displays the last value again (from the *value history*; see Section 8.8 [Value history], page 74). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See Section 8.5 [Examining memory], page 67.

If you are interested in information about types, or about how the fields of a struct or a class are declared, use the `ptype` *`exp`* command rather than `print`. See Chapter 10 [Examining the Symbol Table], page 93.

8.1 Expressions

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

GDB supports array constants in expressions input by the user. The syntax is `{`*element*, *element*. . .`}`. For example, you can use the command `print {1, 2, 3}` to build up an array in memory that calls `malloc` in the target program.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See Chapter 9 [Using GDB with Different Languages], page 79, for information on how to use expressions in other languages.

In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer in order to examine a structure at that address in memory.

GDB supports these operators, in addition to those common to programming languages:

`@` ‘`@`’ is a binary operator for treating parts of memory as arrays. Refer to See Section 8.3 [Artificial arrays], page 65, for more information.

`::` ‘`::`’ allows you to specify a variable in terms of the file or function where it is defined. See Section 8.2 [Program variables], page 64.

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

8.2 Program variables

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

Variables in expressions are understood in the selected stack frame (see Section 6.5 [Selecting a frame], page 53); they must be either:

- global (or file-static)

or

- visible according to the scope rules of the programming language from the point of execution in that frame

This means that in the function

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

you can examine and use the variable `a` whenever your program is executing within the function `foo`, but you can only use or examine the variable `b` while your program is executing inside the block where `b` is declared.

However, you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file, using the colon-colon notation:

```
file::variable
function::variable
```

Here *file* or *function* is the name of the context for the static *variable*. In the case of file names, you can use quotes to make sure GDB parses the file name as a single word. For example, to print a global value of `x` defined in ‘`f2.c`’:

```
((gdb)) p 'f2.c'::x
```

This use of '`::`' is very rarely in conflict with the very similar use of the same notation in C++. GDB also supports use of the C++ scope resolution operator in GDB expressions.

> *Warning:* Occasionally, a local variable may appear to have the wrong value at certain points in a function just after entry to a new scope, and just before exit.

You may see this problem when you are stepping by machine instructions. This is because, on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

This may also happen when the compiler does significant optimizations. To be sure of always seeing accurate values, turn off all optimization when compiling.

Another possible effect of compiler optimizations is to optimize unused variables out of existence, or assign variables to registers (as opposed to memory addresses). Depending on the support for such cases offered by the debug info format used by the compiler, GDB might not be able to display values for such local variables. If that happens, GDB will print a message like this:

```
No symbol "foo" in current context.
```

To solve such problems, either recompile without optimizations, or use a different debug info format, if the compiler supports several such formats. For example, GCC, the GNU C/C++ compiler usually supports the '`-gstabs`' option. The '`-gstabs`' produces debug information in a format that is superior to formats such as COFF. You may be able to use DWARF-2 ('`-gdwarf-2`'), which is also an effective form for debug info. See Section 4.1 [Compiling for Debugging], page 23.

8.3 Artificial arrays

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

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

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

you can print the contents of `array` with

```
p *array@len
```

The left operand of '`@`' must reside in memory. Array values made with '`@`' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see Section 8.8 [Value history], page 74), after printing one out.

Another way to create an artificial array is to use a cast. This re-interprets a value as if it were an array. The value need not be in memory:

```
((gdb)) p/x (short[2])0x12345678
$1 = {0x1234, 0x5678}
```

As a convenience, if you leave the array length out (as in '`(`*`type`*`[])`*`value`*'), GDB calculates the size to fill the value (as '`sizeof(`*`value`*`)/sizeof(`*`type`*`)`':

```
((gdb)) p/x (short[])0x12345678
$2 = {0x1234, 0x5678}
```

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (See (see Section 8.9 [Convenience variables], page 75)) as a counter in an expression that prints the first interesting value, and then repeat that expression via ⟨RET⟩. For instance, suppose you have an array `dtab` of pointers to structures, and you are interested in the values of a field `fv` in each structure. Here is an example of what you might type:

```
set $i = 0
p dtab[$i++]->fv
⟨RET⟩
⟨RET⟩
...
```

8.4 Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

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

`x`	Regard the bits of the value as an integer, and print the integer in hexadecimal.
`d`	Print as integer in signed decimal.
`u`	Print as integer in unsigned decimal.
`o`	Print as integer in octal.
`t`	Print as integer in binary. The letter '`t`' stands for "two".[1]

[1] '`b`' cannot be used because these format letters are also used with the `x` command, where '`b`' stands for "byte"; see Section 8.5 [Examining memory], page 67.

`a` Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
((gdb)) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

`c` Regard as an integer and print it as a character constant.

`f` Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see Section 8.10 [Registers], page 77), type

```
p/x $pc
```

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

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

8.5 Examining memory

You can use the command `x` (for "examine") to examine memory in any of several formats, independent of your program data types.

`x/`*`nfu addr`*
`x` *`addr`*
`x` Use the `x` command to examine memory.

n, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; *addr* is an expression giving the address where you want to start displaying memory. If you use defaults for *nfu*, you need not type the slash '`/`'. Several commands set convenient defaults for *addr*.

n, the repeat count
The repeat count is a decimal integer and the default is 1. It specifies how much memory (counting by units *u*) to display.

f, the display format
The display format is one of the formats used by `print`, '`s`' (null-terminated string), or '`i`' (machine instruction). The default is '`x`' (hexadecimal) initially. The default changes each time you use either `x` or `print`.

u, the unit size
The unit size is any of

`b`	Bytes.
`h`	Halfwords (two bytes).
`w`	Words (four bytes). This is the initial default.

`g` Giant words (eight bytes).

Each time you specify a unit size with `x`, that size becomes the default unit the next time you use `x`. (For the '`s`' and '`i`' formats, the unit size is ignored and is normally not written.)

addr, starting display address

addr is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. Refer to See Section 8.1 [Expressions], page 63, for more information on expressions. The default for *addr* is usually just after the last address examined—but several other commands also set the default address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, '`x/3uh 0x54320`' is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers ('`u`'), starting at address `0x54320`. '`x/4xw $sp`' prints the four words ('`w`') of memory above the stack pointer (here, '`$sp`'; see Section 8.10 [Registers], page 77) in hexadecimal ('`x`').

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you do not have to remember whether unit size or format comes first; either order works. The output specifications '`4xw`' and '`4wx`' mean exactly the same thing. (However, the count *n* must come first; '`wx4`' does not work.)

Even though the unit size *u* is ignored for the formats '`s`' and '`i`', you might still want to use a count *n*; for example, '`3i`' specifies that you want to see three machine instructions, including any operands. The command `disassemble` gives an alternative way of inspecting machine instructions; see Section 7.4 [Source and machine code], page 60.

All the defaults for the arguments to `x` are designed to make it easy to continue scanning memory with minimal specifications each time you use `x`. For example, after you have inspected three machine instructions with '`x/3i addr`', you can inspect the next seven with just '`x/7`'. If you use ⟨RET⟩ to repeat the `x` command, the repeat count *n* is used again; the other arguments default as for successive uses of `x`.

The addresses and contents printed by the `x` command are not saved in the value history because there is often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables `$_` and `$__`. After an `x` command, the last address examined is available for use in expressions in the convenience variable `$_`. The contents of that address, as examined, are available in the convenience variable `$__`.

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

8.6 Automatic display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

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

This display shows item numbers, expressions and their current values. As with displays you request manually using `x` or `print`, you can specify the output format you prefer; in fact, `display` decides whether to use `print` or `x` depending on how elaborate your format specification is—it uses `x` if you specify a unit size, or one of the two formats ('`i`' and '`s`') that are only supported by `x`; otherwise it uses `print`.

`display` ***`expr`***
: Add the expression *expr* to the list of expressions to display each time your program stops. See Section 8.1 [Expressions], page 63.

  `display` does not repeat if you press ⟨RET⟩ again after using it.

`display/`***`fmt expr`***
: For *fmt* specifying only a display format and not a size or count, add the expression *expr* to the auto-display list but arrange to display it each time in the specified format *fmt*. See Section 8.4 [Output formats], page 66.

`display/`***`fmt addr`***
: For *fmt* '`i`' or '`s`', or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing '`x/`***`fmt addr`***'. See Section 8.5 [Examining memory], page 67.

For example, '`display/i $pc`' can be helpful, to view the machine instruction about to be executed each time execution stops ('`$pc`' is a common name for the program counter; see Section 8.10 [Registers], page 77).

`undisplay` ***`dnums`*** `...`
`delete display` ***`dnums`*** `...`
: Remove item numbers *dnums* from the list of expressions to display.

  `undisplay` does not repeat if you press ⟨RET⟩ after using it. (Otherwise you would just get the error '`No display number ...`'.)

`disable display` ***`dnums`*** `...`
: Disable the display of item numbers *dnums*. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

`enable display` ***`dnums`*** `...`
: Enable display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise.

`display`
: Display the current values of the expressions on the list, just as is done when your program stops.

`info display`
: Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which

would not be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables is not defined. For example, if you give the command `display last_char` while inside a function with an argument `last_char`, GDB displays this argument while your program continues to stop inside that function. When it stops elsewhere, where there is no variable `last_char`, the display is disabled automatically. The next time your program stops where `last_char` is meaningful, you can enable the display expression again.

8.7 Print settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

`set print address`
`set print address on`

GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is `on`. For example, this is what a stack frame display looks like with `set print address on`:

```
((gdb)) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote != def_lquote)
```

`set print address off`

Do not print addresses when displaying their contents. For example, this is the same stack frame displayed with `set print address off`:

```
((gdb)) set print addr off
((gdb)) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530         if (lquote != def_lquote)
```

You can use '`set print address off`' to eliminate all machine dependent displays from the GDB interface. For example, with `print address off`, you should get the same text for backtraces on all machines—whether or not they involve pointer arguments.

`show print address`

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest previous symbol plus an offset. If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to clarify it. One way to do this is with `info line`. For example '`info line *0x4537`'. Alternately, you can set GDB to print the source file and the line number when it prints a symbolic address:

`set print symbol-filename on`
: Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

`set print symbol-filename off`
: Do not print source file name and line number of a symbol. This is the default.

`show print symbol-filename`
: Show whether or not GDB will print the source file name and line number of a symbol in the symbolic form of an address.

Another situation where it is helpful to show symbol filenames and line numbers is when disassembling code. GDB shows you the line number and source file that corresponds to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

`set print max-symbolic-offset` *`max-offset`*
: Tell GDB to only display the symbolic form of an address if the offset between the closest symbol and the address is less than *max-offset*. The default is 0, which tells GDB to always print the symbolic form of an address if any symbol precedes it.

`show print max-symbolic-offset`
: Ask how large the maximum offset is that GDB prints in a symbolic address.

If you have a pointer and you are not sure where it points, try '`set print symbol-filename on`'. Then you can determine the name and source file location of the variable where it points, using '`p/a` *`pointer`*'. This interprets the address in symbolic form. For example, here GDB shows that a variable `ptt` points at another variable `t`, defined in '`hi2.c`':

```
((gdb)) set print symbol-filename on
((gdb)) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

> *Warning:* For pointers that point to a local variable, '`p/a`' does not show the symbol name and filename of the referent, even with the appropriate `set print` options turned on.

Other settings to control how different kinds of objects are printed:

`set print array`
`set print array on`
: Pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

`set print array off`
: Return to compressed format for arrays.

`show print array`
: Show whether compressed or pretty format is selected for displaying arrays.

`set print elements` *`number-of-elements`*
: Set a limit on how many elements of an array GDB will print. If GDB is printing a large array, it stops printing after it has printed the number of elements set by the `set print elements` command. This limit also applies to the display of strings. When GDB starts, this limit is set to 200. Setting *number-of-elements* to zero means that the printing is unlimited.

`show print elements`
: Display the number of elements of a large array that GDB will print. If the number is 0, then the printing is unlimited.

`set print null-stop`
: Cause GDB to stop printing the characters of an array when the first NULL is encountered. This is useful when large arrays actually contain only short strings. The default is off.

`set print pretty on`
: Cause GDB to print structures in an indented format with one member per line, like this:

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

`set print pretty off`
: Cause GDB to print structures in a compact format, like this:

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

This is the default format.

`show print pretty`
: Show which format GDB is using to print structures.

`set print sevenbit-strings on`
: Print using only seven-bit characters; if this option is set, GDB displays any eight-bit characters (in strings or character values) using the notation `\`*nnn*. This setting is best if you are working in English (ASCII) and you use the high-order bit of characters as a marker or “meta” bit.

`set print sevenbit-strings off`
: Print full eight-bit characters. This allows the use of more international character sets, and is the default.

`show print sevenbit-strings`
: Show whether or not GDB is printing only seven-bit characters.

`set print union on`
: Tell GDB to print unions which are contained in structures. This is the default setting.

`set print union off`
: Tell GDB not to print unions which are contained in structures.

`show print union`
: Ask GDB whether or not it will print unions which are contained in structures.

For example, given the declarations

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

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

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

with `set print union on` in effect '`p foo`' would print

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

and with `set print union off` in effect it would print

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

These settings are of interest when debugging C++ programs:

`set print demangle`
`set print demangle on`
: Print C++ names in their source form rather than in the encoded ("mangled") form passed to the assembler and linker for type-safe linkage. The default is on.

`show print demangle`
: Show whether C++ names are printed in mangled or demangled form.

`set print asm-demangle`
`set print asm-demangle on`
: Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

`show print asm-demangle`
: Show whether C++ names in assembly listings are printed in mangled or demangled form.

`set demangle-style` *`style`*
: Choose among several encoding schemes used by different compilers to represent C++ names.

On HP-UX, WDB automatically chooses the appropriate style.

The choices for *style* currently supported are:

`auto` Allow GDB to choose a decoding style by inspecting your program.

`gnu` Decode based on the GNU C++ compiler (`g++`) encoding algorithm.

`hp` Decode based on the HP ANSI C++ (`aCC`) encoding algorithm.

`lucid` Decode based on the Lucid C++ compiler (`lcc`) encoding algorithm.

`arm` Decode using the algorithm in the *C++ Annotated Reference Manual*. **Warning:** this setting alone is not sufficient to allow debugging `cfront` generated executables. GDB would require further enhancement to permit that.

If you omit *style*, you will see a list of possible formats.

`show demangle-style`
Display the encoding style currently in use for decoding C++ symbols.

`set print object`
`set print object on`
When displaying a pointer to an object, identify the *actual* (derived) type of the object rather than the *declared* type, using the virtual function table.

`set print object off`
Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

`show print object`
Show whether actual, or declared, object types are displayed.

`set print static-members`
`set print static-members on`
Print static members when displaying a C++ object. The default is on.

`set print static-members off`
Do not print static members when displaying a C++ object.

`show print static-members`
Show whether C++ static members are printed, or not.

`set print vtbl`
`set print vtbl on`
Pretty print C++ virtual function tables. The default is off. (The `vtbl` commands do not work on programs compiled with the HP ANSI C++ compiler (`aCC`).)

`set print vtbl off`
Do not pretty print C++ virtual function tables.

`show print vtbl`
Show whether C++ virtual function tables are pretty printed, or not.

8.8 Value history

Values printed by the `print` command are saved in the GDB *value history*. This allows you to refer to them in other expressions. Values are kept until the symbol table is re-read or discarded (for example with the `file` or `symbol-file` commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given *history numbers* by which you can refer to them. These are a range of integers starting with one. `print` shows you the history number assigned to a value by printing '`$num = `' before the value; here *num* is the history number.

To refer to any previous value, use '`$`' followed by the history number of the value. The way `print` labels its output is designed to remind you of this. Just `$` refers to the most recent value in the history, and `$$` refers to the value before that. `$$n` refers to the *n*th value from the end; `$$2` is the value just prior to `$$`, `$$1` is equivalent to `$$`, and `$$0` is equivalent to `$`.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

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

```
p *$.next
```

You can print successive links in the chain by repeating this command using the ⟨RET⟩ key.

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

```
print x
set x=5
```

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

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

`show values n`
: Print ten history values centered on history item number *n*.

`show values +`
: Print ten history values following the values last printed. If no more values are available, `show values +` produces no display.

Pressing ⟨RET⟩ to repeat `show values n` has exactly the same effect as '`show values +`'.

8.9 Convenience variables

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

Convenience variables are prefixed with '`$`'. Any name preceded by '`$`' can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see Section 8.10 [Registers], page 77). (Value history references, in contrast, are *numbers* preceded by '`$`'. See Section 8.8 [Value history], page 74.)

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

```
set $foo = *object_ptr
```

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

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

Convenience variables have no fixed types. You can assign a convenience variable any type of value, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

`show convenience`
: Print a list of convenience variables used so far, and their values. Abbreviated `show conv`.

A convenient variable can be used as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

```
set $i = 0
print bar[$i++]->contents
```

Repeat that command by typing ⟨RET⟩.

Some convenience variables are created automatically by GDB and assigned values.

`$_`
: The variable `$_` is automatically set by the `x` command to the last address examined (see Section 8.5 [Examining memory], page 67). Other commands which provide a default address for `x` to examine, also set `$_` to that address. These commands include `info line` and `info breakpoint`. The type of `$_` is `void *` except when set by the `x` command, in which case it is a pointer to the type of `$__`.

`$__`
: The variable `$__` is automatically set by the `x` command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

`$_exitcode`
: The variable `$_exitcode` is automatically set to the exit code when the program being debugged terminates.

On HP-UX systems, if you refer to a function or variable name that begins with a dollar sign, GDB searches for a user or system name first, before it searches for a convenience variable.

8.10 Registers

You can refer to machine register contents, in expressions, as variables with names starting with '`$`'. The names of registers are different for each machine. Use `info registers` to view the names used on your machine.

`info registers`
: Print the names and values of all registers except floating-point registers (in the selected stack frame).

`info all-registers`
: Print the names and values of all registers, including floating-point registers.

`info registers` *`regname`* `...`
: Print the *relativized* value of each specified register *regname*. As discussed in detail below, register values are normally relative to the selected stack frame. *regname* may be any register name valid on the machine you are using, with or without the initial '`$`'.

GDB has four standard register names that are available (in expressions) on most machines—whenever they do not conflict with an architecture's canonical mnemonics for registers. The register names `$pc` and `$sp` are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer[2] with

```
set $sp += 4
```

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The `info registers` command shows the canonical names. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`; and on x86-based machines `$ps` is an alias for the EFLAGS register.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can *print* it as a floating point value with '`print/f $`*`regname`*').

Some registers have distinct raw and virtual data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point

[2] This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use `return`; see Section 11.4 [Returning from a function], page 99.

coprocessor are always saved in "extended" (raw) format, but all C programs expect to work with "double" (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the `info registers` command prints the data in both formats.

Normally, register values are relative to the selected stack frame (see Section 6.5 [Selecting a frame], page 53). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with '`frame 0`').

However, GDB must deduce where registers are saved, from the machine code generated by your compiler. If some registers are not saved, or if GDB is unable to locate the saved registers, the selected stack frame makes no difference.

8.11 Printing Floating Point Values

You can print the values of floating-point registers in different formats.

To print both single and double-precision values:

```
(gdb) info reg $fr5
fr5     (single precision)     10.1444092
fr5
```

To get the bit pattern, try the following macro:

```
define pbits
  set *((float *) $sp)=$arg0
  p/x *((int *) $sp)
end
```

This is what the macro produces:

```
(gdb) pbits $fr6
$1 = 0x4082852d
```

8.12 Floating point hardware

Depending on the configuration, GDB may be able to give you more information about the status of the floating point hardware.

`info float`
: Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip. Currently, '`info float`' is supported on the ARM and x86 machines.

9 Using GDB with Different Languages

Although programming languages generally have common aspects, they are rarely expressed in the same manner. For instance, in ANSI C, dereferencing a pointer `p` is accomplished by `*p`, but in Modula-2, it is accomplished by `p^`. Values can also be represented (and displayed) differently. Hex numbers in C appear as '`0x1ae`', while in Modula-2 they appear as '`1AEH`'.

Language-specific information is built into GDB for some languages, allowing you to express operations like the above in the native language of your program, and allowing GDB to output values in a manner consistent with the syntax of the native language. The language you use to build expressions is called the *working language*.

9.1 Switching between source languages

There are two ways to control the working language. You can have GDB set it automatically, or you can select it manually. You can use the `set language` command for either purpose. On startup, GDB sets the default language automatically. The working language is used to determine how expressions are interpreted, how values are printed, etc.

In addition to the working language, every source file that GDB knows about has its own working language. For some object file formats, the compiler might indicate which language a particular source file is in. However, most of the time GDB infers the language from the name of the file. The language of a source file controls whether C++ names are demangled—this way `backtrace` can show each frame appropriately for its own language. There is no way to set the language of a source file from within GDB, but you can set the language associated with a filename extension. See Section 9.2 [Displaying the language], page 80.

This is a common problem when you use a program, such as `cfront` or `f2c`, that generates C but is written in another language. In that case, make the program use `#line` directives in its C output; that way GDB will know the correct language of the source code of the original program, and will display that source code, not the generated C code.

9.1.1 List of filename extensions and languages

If a source file name ends in one of the following extensions, then GDB infers that its language is the one indicated.

'`.c`'	C source file
'`.C`' '`.cc`' '`.cp`' '`.cpp`' '`.cxx`' '`.c++`'	C++ source file

'.f'
'.F'
'.f90' Fortran source file. GDB does not distinguish between Fortran 77 and Fortran 90 files.

'.s'
'.S' Assembler source file. This actually behaves almost like C, but GDB does not skip over function prologues when stepping.

In addition, you may set the language associated with a filename extension. See Section 9.2 [Displaying the language], page 80.

9.1.2 Setting the working language

If you allow GDB to set the language automatically, expressions are interpreted the same way in your debugging session and your program.

If you wish, you may set the language manually. To do this, issue the command '`set language` *lang*', where *lang* is the name of a language, such as `c`. For a list of the supported languages, type '`set language`'.

Setting the language manually prevents GDB from updating the working language automatically. This can lead to confusion if you try to debug a program when the working language is not the same as the source language, when an expression is acceptable to both languages—but means different things. For instance, if the current source file were written in C, and GDB was parsing Modula-2, a command such as:

```
print a = b + c
```

might not have the effect you intended. In C, this means to add `b` and `c` and place the result in `a`. The result printed would be the value of `a`. In Modula-2, this means to compare `a` to the result of `b+c`, yielding a `BOOLEAN` value.

9.1.3 Having GDB infer the source language

To have GDB set the working language automatically, use '`set language local`' or '`set language auto`'. GDB then infers the working language. That is, when your program stops in a frame (usually by encountering a breakpoint), GDB sets the working language to the language recorded for the function in that frame. If the language for a frame is unknown (that is, if the function or block corresponding to the frame was defined in a source file that does not have a recognized extension), the current working language is not changed, and GDB issues a warning.

This may not seem necessary for most programs, which are written entirely in one source language. However, program modules and libraries written in one source language can be used by a main program written in a different source language. Using '`set language auto`' in this case frees you from having to set the working language manually.

9.2 Displaying the language

The following commands help you find out which language is the working language, and also what language source files were written in.

`show language`
: Display the current working language. This is the language you can use with commands such as `print` to build and compute expressions that may involve variables in your program.

`info frame`
: Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See Section 6.6 [Information about a frame], page 54, to identify the other information listed here.

`info source`
: Display the source language of this source file. Refer to See Chapter 10 [Examining the Symbol Table], page 93, to identify the other information listed here.

In unusual circumstances, you may have source files with extensions not in the standard list. You can then set the extension associated with a language explicitly:

`set extension-language .ext language`
: Set source files with extension *.ext* to be assumed to be in the source language *language*. However, this is not valid on Unix systems.

`info extensions`
: List all the filename extensions and the associated languages. Not valid on Unix systems.

9.3 Type and range checking

Some languages are designed to guard you against making seemingly common errors through a series of compile and run-time checks. These include checking the type of arguments to functions and operators, and making sure mathematical overflows are caught at run time. Checks such as these help to ensure the correctness of the program once it has been compiled by eliminating type mismatches, and providing active checks for range errors when your program is running.

GDB can check for conditions like the above if you wish. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation via the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your source language. See Section 9.4 [Supported languages], page 83, for the default settings of supported languages.

9.3.1 An overview of type checking

Some languages are strongly typed, meaning that the arguments to operators and functions have to be of the correct type, otherwise an error occurs. These checks prevent type mismatch errors from causing run-time problems. For example,

```
1 + 2 ⇒ 3
```

but

```
error  1 + 2.3
```

The second example fails because the `CARDINAL` 1 is not type-compatible with the `REAL` 2.3.

For the expressions you use in GDB commands, you can tell the GDB type checker to skip checking; to treat any mismatches as errors and abandon the expression; or to only issue warnings when type mismatches occur, and evaluate the expression anyway. When you choose the last of these, GDB evaluates expressions like the second example above, but also issues a warning.

Even if you turn type checking off, there may be other reasons related to type that prevent GDB from evaluating an expression. For instance, GDB does not know how to add an `int` and a `struct foo`. These particular type errors have nothing to do with the language in use, and usually arise from expressions, such as the one described above, which make little sense to evaluate anyway.

Each language defines to what degree it is strict about type. For instance C requires the arguments to arithmetical operators to be numbers. In C, enumerated types and pointers can be represented as numbers, so that they are valid arguments to mathematical operators. See Section 9.4 [Supported languages], page 83, for further details on specific languages.

GDB provides some additional commands for controlling the type checker:

`set check type auto`
: Set type checking on or off based on the current working language. See Section 9.4 [Supported languages], page 83, for the default settings for each language.

`set check type on`
`set check type off`
: Set type checking on or off, overriding the default setting for the current working language. Issue a warning if the setting does not match the language default. If any type mismatches occur in evaluating an expression while type checking is on, GDB prints a message and aborts evaluation of the expression.

`set check type warn`
: Cause the type checker to issue warnings, but to always attempt to evaluate the expression. Evaluating the expression may still be impossible for other reasons. For example, GDB cannot add numbers and structures.

`show type`
: Show the current setting of the type checker, and whether or not GDB is setting it automatically.

9.3.2 An overview of range checking

In some languages it is an error to exceed the bounds of a type; this is enforced with run-time checks. Such range checking is meant to ensure program correctness by making sure computations do not overflow, or indices on an array element access do not exceed the bounds of the array.

For expressions you use in GDB commands, you can tell GDB to treat range errors in one of three ways: ignore them, always treat them as errors and abandon the expression, or issue warnings but evaluate the expression anyway.

A range error can result from numerical overflow, from exceeding an array index bound, or when you type a constant that is not a member of any type. Some languages, however, do not treat overflows as an error. In many implementations of C, mathematical overflow causes the result to "wrap around" to lower values—for example, if *m* is the largest integer value, and *s* is the smallest, then

```
m + 1 ⇒ s
```

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. Refer to See Section 9.4 [Supported languages], page 83, for further details on specific languages.

GDB provides the following additional commands for controlling the range checker:

`set check range auto`
: Set range checking on or off based on the current working language. See Section 9.4 [Supported languages], page 83, for the default settings for each language.

`set check range on`
`set check range off`
: Set range checking on or off, overriding the default setting for the current working language. A warning is issued if the setting does not match the default language. If a range error occurs and range checking is on, then a message is printed and evaluation of the expression is aborted.

`set check range warn`
: Output messages when the GDB range checker detects a range error, but attempt to evaluate the expression anyway. Evaluating the expression may still be impossible for other reasons, such as accessing memory that the process does not own (a typical example from many Unix systems).

`show range`
: Show the current setting of the range checker, and whether or not it is being set automatically by GDB.

9.4 Supported languages

GDB supports C, C++, and Fortran. Refer to Section 9.4.2 [Fortran], page 90 for specific information about Fortran.

Some GDB features may be used in expressions regardless of the language you use: the GDB `@` and `::` operators, and the '`{type}addr`' construct (see Section 8.1 [Expressions], page 63) can be used with the constructs of any supported language.

The following section discusses GDB support for each source language. These sections are not meant to be language tutorials or references, but serve only as a reference guide to what the GDB expression parser accepts, and what input and output formats should look like for different languages.

9.4.1 C and C++

Since C and C++ are so closely related, many features of GDB apply to both languages. Whenever this is the case, we discuss those languages together.

The C++ debugging facilities are jointly implemented by the C++ compiler and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with a supported C++ compiler, such as GNU `g++`, or the HP ANSI C++ compiler (`aCC`).

For best results when using GNU C++, use the stabs debugging format. You can select that format explicitly with the `g++` command-line options '`-gstabs`' or '`-gstabs+`'. Refer to section "Options for Debugging Your Program or GNU CC" in *Using* GNU *CC*, for more information.

9.4.1.1 C and C++ operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types.

For the purposes of C and C++, the following definitions hold:

- *Integral types* include `int` with any of its storage-class specifiers; `char`; `enum`; and, for C++, `bool`.
- *Floating-point types* include `float`, `double`, and `long double` (if supported by the target platform).
- *Pointer types* include all types defined as `(type *)`.
- *Scalar types* include all of the above.

The following operators are supported. They are listed here in order of increasing precedence:

`,` The comma or sequencing operator. Expressions in a comma-separated list are evaluated from left to right, with the result of the entire expression being the last expression evaluated.

`=` Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.

`op``=` Used in an expression of the form `a op= b`, and translated to `a = a op b`. *`op`*`=` and `=` have the same precedence. *op* is any one of the operators `|`, `^`, `&`, `<<`, `>>`, `+`, `-`, `*`, `/`, `%`.

`?:` The ternary operator. `a ? b : c` can be thought of as: if *a* then *b* else *c*. *a* should be of an integral type.

`||` Logical OR. Defined on integral types.

`&&` Logical AND. Defined on integral types.

`|` Bitwise OR. Defined on integral types.

`^` Bitwise exclusive-OR. Defined on integral types.

`&` Bitwise AND. Defined on integral types.

`==, !=` Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<, >, <=, >=`
Less than, greater than, less than or equal, greater than or equal. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<<, >>` left shift, and right shift. Defined on integral types.

`@` The GDB "artificial array" operator (see Section 8.1 [Expressions], page 63).

`+, -` Addition and subtraction. Defined on integral types, floating-point types and pointer types.

`*, /, %` Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.

`++, --` Increment and decrement. When appearing before a variable, the operation is performed before the variable is used in an expression; when appearing after it, the value of the variable is used before the operation takes place.

`*` Pointer dereferencing. Defined on pointer types. Same precedence as `++`.

`&` Address operator. Defined on variables. Same precedence as `++`.

For debugging C++, GDB implements a use of '`&`' beyond what is allowed in the C++ language itself: you can use '`&(&`*ref*`)`' (or, if you prefer, simply '`&&`*ref*') to examine the address where a C++ reference variable (declared with '`&`*ref*') is stored.

`-` Negative. Defined on integral and floating-point types. Same precedence as `++`.

`!` Logical negation. Defined on integral types. Same precedence as `++`.

`~` Bitwise complement operator. Defined on integral types. Same precedence as `++`.

`., ->` Structure member, and pointer-to-structure member. For convenience, GDB regards the two as equivalent, choosing whether to dereference a pointer based on the stored type information. Defined on `struct` and `union` data.

`.*, ->*` Dereferences pointers to members.

`[]` Array indexing. `a[`*i*`]` is defined as `*(a+`*i*`)`. Same precedence as `->`.

`()` Function parameter list. Same precedence as `->`.

`::` C++ scope resolution operator. Defined on `struct`, `union`, and `class` types.

`::` Double colons also represent the GDB scope operator (see Section 8.1 [Expressions], page 63). Same precedence as `::`, above.

If an operator is redefined in the user code, GDB usually attempts to invoke the redefined version instead of using the original meaning.

9.4.1.2 C and C++ constants

GDB allows you to express the constants of C and C++ in the following ways:

- Integer constants are a sequence of digits. Octal constants are specified by a leading '`0`' (i.e. zero), and hexadecimal constants by a leading '`0x`' or '`0X`'. Constants can also end with a letter '`l`', specifying that the constant should be treated as a `long` value.
- Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: '`e[[+]|-]`*nnn*', where *nnn* is a sequence of digits. The '`+`' is optional for positive exponents. A floating-point constant may also end with a letter '`f`' or '`F`', specifying that the constant should be treated as being of the `float` (as opposed to the default `double`) type; or with a letter '`l`' or '`L`', which specifies a `long double` constant.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (`'`), or a number or the ordinal value of the corresponding character (usually its ASCII value). Within quotes, the single character may be represented by a letter or by *escape sequences*, which are of the form '`\`*nnn*', where *nnn* is the octal representation of the character's ordinal value; or of the form '`\x`', where '`x`' is a predefined special character—for example, '`\n`' for newline.
- String constants are a sequence of character constants surrounded by double quotes (`"`). Any valid character constant (as described above) may appear. Double quotes within the string must be preceded by a backslash, so for instance '`"a\"b'c"`' is a string of five characters.
- Pointer constants are an integral value. You can also write pointers to constants using the C operator '`&`'.
- Array constants are comma-separated lists surrounded by braces '`{`' and '`}`'; for example, '`{1,2,3}`' is a three-element array of integers, '`{{1,2}, {3,4}, {5,6}}`' is a three-by-two array, and '`{&"hi", &"there", &"fred"}`' is a three-element array of pointers.

9.4.1.3 C++ expressions

GDB expression handling can interpret most C++ expressions.

> *Warning:* GDB can only debug C++ code if you use the proper compiler. Typically, C++ debugging depends on the use of additional debugging information in the symbol table, and thus requires special support. In particular, if your compiler generates a.out, MIPS ECOFF, RS/6000 XCOFF, or ELF with stabs extensions to the symbol table, these facilities are all available. (With GNU CC, you can use the '`-gstabs`' option to request stabs debugging extensions explicitly.) Where the object code format is standard COFF or DWARF in ELF, on the other hand, most of the C++ support in GDB does *not* work.

1. Member function calls are allowed; you can use expressions like

```
count = aml->GetOriginal(x, y)
```

2. While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer `this` following the same rules as C++.
3. You can call overloaded functions; GDB resolves the function call to the right definition, with some restrictions. GDB does not perform overload resolution involving user-defined type conversions, calls to constructors, or instantiations of templates that do not exist in the program. It also cannot handle ellipsis argument lists or default arguments.

   It does perform integral conversions and promotions, floating-point promotions, arithmetic conversions, pointer conversions, conversions of class objects to base classes, and standard conversions such as those of functions or arrays to pointers; it requires an exact match on the number of function arguments.

   Overload resolution is always performed, unless you have specified `set overload-resolution off`. See Section 9.4.1.7 [GDB features for C++], page 88.

   You must specify `set overload-resolution off` in order to use an explicit function signature to call an overloaded function, as in

   ```
   p 'foo(char,int)'('x', 13)
   ```

   The GDB command-completion facility can simplify this. Refer to Section 3.2 [Command completion], page 17.
4. GDB understands variables declared as C++ references; you can use them in expressions just as you do in C++ source—they are automatically dereferenced.

   In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The *address* of a reference variable is always shown, unless you have specified '`set print address off`'.
5. GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in an expression like '*`scope1`*`::`*`scope2`*`::`*`name`*'. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see Section 8.2 [Program variables], page 64).

In addition, when used with the HP aC++ compiler, GDB supports calling virtual functions correctly, printing out virtual bases of objects, calling functions in a base subobject, casting objects, and invoking user-defined operators.

> *Note:* GDB cannot display debugging information for classes or functions defined in a shared library that is not compiled for debugging (with the `-g0` option). GDB displays the function with the message `<no data fields>`.
>
> For example, after '`d3`' is created by the following line:
>
> '`RWCollectableDate d3(15,5,2001);`'
>
> printing the variable or class returns:
>
> ```
> (gdb) p d3
> $3 = {<No data fields>}
> (gdb) ptype RWCollectableDate
> type = class RWCollectableDate {
>     <no data fields>
> ```

9.4.1.4 C and C++ defaults

If you allow GDB to set type and range checking automatically, they both default to `off` whenever the working language changes to C or C++. This happens regardless of whether you or GDB selects the working language.

If you allow GDB to set the language automatically, it recognizes source files whose names end with '`.c`', '`.C`', or '`.cc`', etc, and when GDB enters code compiled from one of these files, it sets the working language to C or C++. Refer to See Section 9.1.3 [Having GDB infer the source language], page 80, for further details.

9.4.1.5 C and C++ type and range checks

By default, when GDB parses C or C++ expressions, type checking is not used. However, if you turn type checking on, GDB considers two variable types equivalent if:

- The two variables are structured and have the same structure, union, or enumerated tag.
- The two variables have the same type name, or types that have been declared equivalent through `typedef`.

Range checking, if turned on, is done on mathematical operations. Array indices are not checked, since they are often used to index a pointer that is not itself an array.

9.4.1.6 GDB and C

The `set print union` and `show print union` commands apply to the `union` type. When set to '`on`', any `union` that is inside a `struct` or `class` is also printed. Otherwise, it appears as '`{...}`'.

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See Section 8.1 [Expressions], page 63.

9.4.1.7 GDB features for C++

Some GDB commands are particularly useful with C++, and some are designed specifically for use with C++. Here is a summary:

breakpoint menus

When you want a breakpoint in a function whose name is overloaded, GDB breakpoint menus help you specify which function definition you want. See Section 5.1.7 [Breakpoint menus], page 42.

`rbreak` *`regex`*

Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. See Section 5.1.1 [Setting breakpoints], page 33.

`catch throw`
`catch catch`
: Debug C++ exception handling using these commands. See Section 5.1.2 [Setting catchpoints], page 37.

`ptype` *`typename`*
: Print inheritance relationships as well as other information for type. *typename*. See Chapter 10 [Examining the Symbol Table], page 93.

`set print demangle`
`show print demangle`
`set print asm-demangle`
`show print asm-demangle`
: Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. See Section 8.7 [Print settings], page 70.

`set print object`
`show print object`
: Choose whether to print derived (actual) or declared types of objects. See Section 8.7 [Print settings], page 70.

`set print vtbl`
`show print vtbl`
: Control the format for printing virtual function tables. See Section 8.7 [Print settings], page 70. (The `vtbl` commands do not work on programs compiled with the HP ANSI C++ compiler (`aCC`).)

`set overload-resolution on`
: Enable overload resolution for C++ expression evaluation. The default is on. For overloaded functions, GDB evaluates the arguments and searches for a function whose signature matches the argument types, using the standard C++ conversion rules (see Section 9.4.1.3 [C++ expressions], page 86, for details). If it cannot find a match, it emits a message.

`set overload-resolution off`
: Disable overload resolution for C++ expression evaluation. For overloaded functions that are not class member functions, GDB chooses the first function of the specified name that it finds in the symbol table, whether or not its arguments are of the correct type. For overloaded functions that are class member functions, GDB searches for a function whose signature *exactly* matches the argument types.

`show overload-resolution`
: Display current overload resolution setting for C++ expression evaluation.

Overloaded symbol names
: You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type *`symbol`*`(`*`types`*`)` rather than just *symbol*. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See Section 3.2 [Command completion], page 17, for details on how to do this.

9.4.2 Fortran

You can use WDB to debug programs written in Fortran. WDB does not distinguish between Fortran 77 and Fortran 90 files.

WDB provides the following command to control case sensitivity:

`case-sensitive [on | off]`
: The default for Fortran is off, while for other languages the default is on.

Other supported features are:

- Fortran 90 pointers
- Structures and unions
- Calling functions with integer, logical, real, complex arguments
- Intrinsic support

9.4.2.1 Fortran types

Fortran types supported:

`integer*1, integer*2, integer*4, integer*8`
`logical*1, logical*2, logical*4, logical*8`
`byte`

`real*4, real*8, real*16`
`complex*8, complex*16`
`character*len, character*(*) [len is a user supplied length]`
`arrays`

- allocatable
- assumed-size
- assumed-shape
- adjustable
- automatic
- explicit-shape

Array elements are displayed in column-major order. Use () for array member access (e.g, arr(i) instead of arr[i]). Use `set print elements` to control the number of elements printed out when specifying a whole array. The default is 200 elements or the number of elements of the array, which ever is smaller.

9.4.2.2 Fortran operators

The following Fortran operators are listed here in the order of increasing precedence:

`=` Assignment

`*, -, *, /`
: Binary operators

`+, -`	Unary operators
`**`	Exponentiation
`.EQ., =`	Equal
`.NE., /=`	Not equal, or concatenation
`.LT., <`	Less than
`.LE., <=`	Less than or equal to
`.GT., >`	Greater than
`.GE., >=`	Greater than or equal to
`//`	Concatenation
`.NOT.`	Logical negation
`.AND.`	Logical AND
`.OR.`	Logical OR
`.EQV.`	Logical equivalence
`.NEQV., .XOR.`	Logical non-equivalence

Logical constants are represented as .TRUE. or .FALSE.

GDB includes support for viewing Fortran common blocks.

`info common`
: Lists common blocks visible in the current frame.

`info common <common_block_name>`
: Lists values of variables in the named common block.

Fortran entry points are supported.

You can set a break point specifying an entry point name.

9.4.2.3 Fortran special issues

Fortran allows `main` to be a non-`main` procedure; therefore, to set a breakpoint in the main program, use `break _MAIN_` or `break <`*`program_name`*`>`.

Do not use `break main` unless it is the name of a non-`main` procedure.

10 Examining the Symbol Table

The commands described in this chapter allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB (see Section 2.1.1 [Choosing files], page 12), or by one of the file-management commands (see Section 12.1 [Commands to specify files], page 103).

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see Section 8.2 [Program variables], page 64). File names are recorded in object files as debugging symbols, but GDB would ordinarily parse a typical file name, like '`foo.c`', as the three words '`foo`' '`.`' '`c`'. To allow GDB to recognize '`foo.c`' as a single symbol, enclose it in single quotes; for example,

```
p 'foo.c'::x
```

looks up the value of `x` in the scope of the file '`foo.c`'.

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

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

`whatis` *`expr`*
: Print the data type of expression *expr*. *expr* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See Section 8.1 [Expressions], page 63.

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

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

`ptype` *`expr`*
`ptype`
: Print a description of the type of expression *expr*. `ptype` differs from `whatis` by printing a detailed description, instead of just the name of the type.

  For example, for this variable declaration:

  ```
  struct complex {double real; double imag;} v;
  ```

  the two commands give this output:

```
((gdb)) whatis v
type = struct complex
((gdb)) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with `whatis`, using `ptype` without an argument refers to the type of `$`, the last value in the value history.

`info types` *regexp*
`info types`
: Print a brief description of all types whose names match *regexp* (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, '`i type value`' gives information on all types in your program whose names include the string `value`, but '`i type ^value$`' gives information only on types whose complete name is `value`.

  This command differs from `ptype` in two ways: first, like `whatis`, it does not print a detailed description; second, it lists all source files where a type is defined.

`info source`
: Show the name of the current source file—that is, the source file for the function containing the current point of execution—and the language it was written in.

`info sources`
: Print the names of all source files in your program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols will be read when needed.

`info functions`
: Print the names and data types of all defined functions.

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

`info variables`
: Print the names and data types of all variables that are declared outside of functions (i.e., excluding local variables).

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

  Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. For example, in VxWorks you can simply recompile a defective object file and keep on running. If you are running on one of these systems, you can allow GDB to reload the symbols for automatically relinked modules:

`set symbol-reloading on`
: Replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.

`set symbol-reloading off`
: Do not replace symbol definitions when encountering object files of the same name more than once. This is the default state; if you are not running on a system that permits automatic relinking of modules, you should leave `symbol-reloading` off, since otherwise GDB may discard symbols when linking large programs, that may contain several modules (from different directories or libraries) with the same name.

`show symbol-reloading`
: Show the current `on` or `off` setting.

`set opaque-type-resolution on`
: Tell GDB to resolve opaque types. An opaque type is a type declared as a pointer to a `struct`, `class`, or `union`—for example, `struct MyType *`—that is used in one source file although the full declaration of `struct MyType` is in another source file. The default is on.

  A change in the setting of this subcommand will not take effect until the next time symbols for a file are loaded.

`set opaque-type-resolution off`
: Tell GDB not to resolve opaque types. In this case, the type is printed as follows:

```
{<no data fields>}
```

`show opaque-type-resolution`
: Show whether opaque types are resolved or not.

`maint print symbols` *`filename`*
`maint print psymbols` *`filename`*
`maint print msymbols` *`filename`*
: Write a dump of debugging symbol data into the file *filename*. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use '`maint print symbols`', GDB includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols GDB has read. You can use the command `info sources` to find out which files these are. If you use '`maint print psymbols`' instead, the dump shows information about symbols that GDB only knows partially—that is, symbols defined in files that GDB has skimmed, but not yet read completely. Finally, '`maint print msymbols`' dumps just the minimal symbol information required for each object file from which GDB has read some symbols. See Section 12.1 [Commands to specify files], page 103, for a discussion of how GDB reads symbols (in the description of `symbol-file`).

11 Altering Execution

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

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

11.1 Assignment to variables

To alter the value of a variable, evaluate an assignment expression. See Section 8.1 [Expressions], page 63. For example,

```
print x=4
```

stores the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4). See Chapter 9 [Using GDB with Different Languages], page 79, for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the expression's value is not printed and is not put in the value history (see Section 8.8 [Value history], page 74). The expression is evaluated only for its effects.

The `set` command has a number of subcommands that conflict with the names of program variables. The `set variable` command is a better alternative for setting program variables. The following two examples illustrate the same:

- Example 1

  ```
  ((gdb)) whatis width
  type = double
  ((gdb)) p width
  $4 = 13
  ((gdb)) set width=47
  Invalid syntax in expression.
  ```

  The invalid expression, of course, is '`=47`'. In order to actually set the program's variable `width`, use

- Example 2

  ```
  ((gdb)) set var width=47
  ```

  if your program has a variable `g`, you run into problems if you try to set a new value with just '`set g=4`', because GDB has the command `set gnutarget`, abbreviated `set g`:

```
((gdb)) whatis g
type = double
((gdb)) p g
$1 = 1
((gdb)) set g=4
((gdb)) p g
$2 = 1
((gdb)) r

The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/smith/cc_progs/a.out
"/home/smith/cc_progs/a.out": can't open to read symbols:
                                 Invalid bfd target.
((gdb)) show g
The current BFD target is "=4".
```

The steps shown above sets the `gnutarget` to an invalid value in place of the program variable `g`.

In order to set the variable `g`, use

```
((gdb)) set var g=4
```

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

To store values into arbitrary places in memory, use the '`{...}`' construct to generate a value of specified type at a specified address (see Section 8.1 [Expressions], page 63). For example, `{int}0x83040` refers to memory location `0x83040` as an integer (which implies a certain size and representation in memory), and

```
set {int}0x83040 = 4
```

stores the value 4 into that memory location.

11.2 Continuing at a different address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the `continue` command. You can continue at a selected address using one of the following commands:

`jump` *`linespec`*

Resume execution at line *linespec*. Execution stops again immediately if there is a breakpoint there. See Section 7.1 [Printing source lines], page 57, for a description of the different forms of *linespec*. It is common practice to use the `tbreak` command in conjunction with `jump`. See Section 5.1.1 [Setting breakpoints], page 33.

The `jump` command does not change the current stack frame, the stack pointer, the contents of any memory location or any register other than the program counter. If line *linespec* is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the `jump` command requests confirmation if the specified line is not in the function currently executing. However, even bizarre results are predictable if you are well acquainted with the machine-language code of your program.

`jump *address`
: Resume execution at the instruction at address *address*.

On many systems, you can get much the same effect as the `jump` command by storing a new value into the register `$pc`. This does not start the execution of your program at the specified address, instead only changes the program counter.

For example,

```
set $pc = 0x485
```

makes the next `continue` command or stepping command execute at address `0x485`, rather than at the address where your program stopped. See Section 5.2 [Continuing and stepping], page 44.

The most common occasion to use the `jump` command is to back up—perhaps with more breakpoints set—over a portion of a program that has already executed, in order to examine its execution in more detail.

11.3 Giving your program a signal

You can use the following command to send signals to your program:

`signal signal`
: Resume execution where your program stopped, but immediately give it the signal *signal*. *signal* can be the name or the number of a signal. For example, on many systems `signal 2` and `signal SIGINT` are both ways of sending an interrupt signal.

  Alternatively, if *signal* is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinary see the signal when resumed with the `continue` command; '`signal 0`' causes it to resume without a signal.

  `signal` does not repeat when you press ⟨RET⟩ a second time after executing the command.

Invoking the `signal` command is not the same as invoking the `kill` utility from the shell. Sending a signal with `kill` causes GDB to decide what to do with the signal depending on the signal handling tables (see Section 5.3 [Signals], page 46). The `signal` command passes the signal directly to your program.

11.4 Returning from a function

You can use the following command to return from a function:

`return`
`return` *expression*

You can cancel execution of a function call with the `return` command. If you give an *expression* argument, its value is used as the return value from the function value.

When you use `return`, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to `return`.

This pops the selected stack frame (see Section 6.5 [Selecting a frame], page 53), and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The `return` command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the `finish` command (see Section 5.2 [Continuing and stepping], page 44) resumes execution until the selected stack frame returns naturally.

11.5 Calling program functions

`call` *expr*

Evaluate the expression *expr* without displaying `void` returned values.

You can use this variant of the `print` command if you want to execute a function from your program, but without cluttering the output with `void` returned values. If the result is not void, it is printed and saved in the value history.

For the A29K, a user-controlled variable `call_scratch_address` specifies the location of a scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack does not work in systems that have separate instruction and data spaces.

11.6 Patching programs

By default, GDB opens the file containing the executable code of your program (or the corefile) as read-only. This prevents accidental alteration to machine code; and it also prevents you from intentionally patching your program binary.

If you would like to be able to patch the binary, you can specify that explicitly with the `set write` command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

`set write on`
`set write off`

If you specify '`set write on`', GDB opens executable and core files for both reading and writing; if you specify '`set write off`' (the default), GDB opens them as read-only.

If you have already loaded a file, you must load it again (using the `exec-file` or `core-file` command) after changing `set write`, for your new setting to take effect.

`show write`

Display whether executable files and core files are opened for writing as well as reading.

12 GDB Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, you must also tell GDB the name of the core dump file.

12.1 Commands to specify files

You can specify executable and core dump file names as arguments to the GDB start-up command. (see Chapter 2 [Getting In and Out of GDB], page 11).

Occasionally it is necessary to change to a different file during a GDB session. In these situations the GDB commands to specify new files are useful.

`file` *`filename`*
: Use *filename* as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the `run` command. If you do not specify a directory and the file is not found in the GDB working directory, GDB uses the environment variable `PATH` as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the `path` command.

  On systems with memory-mapped files, an auxiliary file named '*`filename`*`.syms`' may hold symbol table information for *filename*. If so, GDB maps in the symbol table from '*`filename`*`.syms`', starting up more quickly. See the descriptions of the file options '`-mapped`' and '`-readnow`' (available on the command line, and with the commands `file`, `symbol-file`, or `add-symbol-file`, described below) for more information.

`file`
: `file` with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file` [*`filename`*]
: Specify that the program to be run (but not the symbol table) is found in *filename*. GDB searches the environment variable `PATH` if necessary to locate your program. Omitting *filename* means to discard information on the executable file.

`symbol-file` [*`filename`*]
: Read symbol table information from file *filename*. `PATH` is searched when necessary. Use the `file` command to get both symbol table and program to run from the same file.

  `symbol-file` with no argument clears out GDB information on the symbol table of your program.

  The `symbol-file` command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

`symbol-file` does not repeat if you press ⟨RET⟩ again after executing it once.

When GDB is configured for a particular environment, it understands debugging information in whatever format is the standard generated for that environment; you may use either a GNU compiler, or other compilers that adhere to the local conventions.

For most kinds of object files, the `symbol-file` command does not normally read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, as they are needed.

The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The `set verbose` command can turn these pauses into messages if desired. See Section 17.6 [Optional warnings and messages], page 252.)

`symbol-file` ***`filename`*** [`-readnow`] [`-mapped`]
`file` ***`filename`*** [`-readnow`] [`-mapped`]

You can override the GDB two-stage strategy for reading symbol tables by using the '`-readnow`' option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available.

If memory-mapped files are available on your system through the `mmap` system call, you can use another option, '`-mapped`', to cause GDB to write the symbols for your program into a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file (if the program has not changed), rather than spending time reading the symbol table from the executable program. Using the '`-mapped`' option has the same effect as starting GDB with the '`-mapped`' command-line option.

You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program.

The auxiliary symbol file for a program called *myprog* is called '*`myprog`*`.syms`'. Once this file exists (so long as it is newer than the corresponding executable), GDB always attempts to use it when you debug *myprog*; no special options or commands are needed.

The '`.syms`' file is specific to the host machine where you run GDB. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

`core-file` [***`filename`***]

Specify the whereabouts of a core dump file to be used as the "contents of memory". Traditionally, core files contain only some parts of the address space of the process that generated them; GDB can access the executable file itself for other parts.

`core-file` with no argument specifies that no core file is to be used.

Note that the core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running.

To do this, use the `kill` command (see Section 4.8 [Killing the child process], page 28).

`add-symbol-file` *`filename address`*
`add-symbol-file` *`filename address`* [`-readnow`] [`-mapped`]
`add-symbol-file` *`filename address data_address bss_address`*
`add-symbol-file` *`filename`* -s*`section address`*

The `add-symbol-file` command reads additional symbol table information from the file *filename*. You would use this command when *filename* has been dynamically loaded (by some other means) into the program that is running. *address* should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify up to three addresses, in which case they are taken to be the addresses of the text, data, and bss segments respectively. For complicated cases, you can specify an arbitrary number of -s*section address* pairs, to give an explicit section name and base address for that section. You can specify any *address* as an expression.

The symbol table of the file *filename* is added to the symbol table originally read with the `symbol-file` command. You can use the `add-symbol-file` command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `symbol-file` command without any arguments.

`add-symbol-file` does not repeat if you press ⟨RET⟩ after using it.

You can use the '`-mapped`' and '`-readnow`' options just as with the `symbol-file` command, to change how GDB manages the symbol table information for *filename*.

`section` The `section` command changes the base address of section SECTION of the exec file to ADDR. This can be used if the exec file does not contain section addresses, (such as in the `a.out` format), or when the addresses specified in the file itself are wrong. Each section must be changed separately. The `info files` command, described below, lists all the sections and their addresses.

`info files`
`info target`

`info files` and `info target` are synonymous; both print the current target (see Chapter 13 [Specifying a Debugging Target], page 109), including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command `help target` lists all possible targets rather than current ones.

All file-specifying commands allow both absolute and relative file names as arguments. GDB always converts the file name to an absolute file name and remembers it that way.

GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library, however—unless you are debugging a core file).

On HP-UX, if the program loads a library explicitly, GDB automatically loads the symbols at the time of the `shl_load` call. See Section 5.1 [Stopping and starting in shared libraries], page 33, for more information.

`info share`
`info sharedlibrary`
: Print the names of the shared libraries which are currently loaded.

`sharedlibrary` *`regex`*
`share` *`regex`*
: Load shared object library symbols for files matching a Unix regular expression. As with files loaded automatically, it only loads shared libraries required by your program for a core file or after typing `run`. If *regex* is omitted all shared libraries required by your program are loaded.

On HP-UX systems, GDB detects the loading of a shared library and automatically reads in symbols from the newly loaded library, up to a threshold that is initially set but that you can modify if you wish.

Beyond that threshold, symbols from shared libraries must be explicitly loaded. To load these symbols, use the command `sharedlibrary` *`filename`*. The base address of the shared library is determined automatically by GDB and need not be specified.

To display or set the threshold, use the commands:

`set auto-solib-add` *`threshold`*
: Set the autoloading size threshold, in megabytes. If *threshold* is nonzero, symbols from all shared object libraries will be loaded automatically when the inferior begins execution or when the dynamic linker informs GDB that a new library has been loaded, until the symbol table of the program and libraries exceeds this threshold. Otherwise, symbols must be loaded manually, using the `sharedlibrary` command. The default threshold is 100 megabytes.

`show auto-solib-add`
: Display the current autoloading size threshold, in megabytes.

12.2 Specifying shared library locations

On HP-UX, when the shared libraries your program uses are in a different directory than the path specified in the source or object files, specify the correct files to use with one of two environment variables.

‘`GDB_SHLIB_PATH`’
: Set this variable to a colon-separated list of directory path names where the desired shared libraries reside. GDB searches specified list of directories for shared libraries before searching the default system directories.

‘`GDB_SHLIB_ROOT`’
: Set this variable to point to the root of the library in which the desired libraries reside.

> *Note:* If you set both the ‘`GDB_SHLIB_PATH`’ and ‘`GDB_SHLIB_ROOT`’ environment variables, the ‘`GDB_SHLIB_PATH`’ behavior overrides ‘`GDB_SHLIB_ROOT`’.

These environment variables are useful when you are analyzing core files on a system other than the one that produced the core file.

For example, if you want GDB to search for libraries in '`/home/debugger/lib`' and '`/tmp/lib`' before searching the default system directories for libraries, you can use this setting:

```
GDB_SHLIB_PATH=/home/debugger/lib:/tmp/lib
```

With this setting, GDB searches the directories in the order specified until it finds a library with the correct name.

In this example, if GDB encounters a library by the name of '`/usr/lib/libsubs.sl`', GDB searches first for '`/home/debugger/lib/libsubs.sl`' and then for '`/tmp/lib/libsubs.sl`'. If neither of these exists, then GDB searches the default system directories and finds '`/usr/lib/libsubs.sl`'.

In most cases, '`GDB_SHLIB_PATH`' allows more flexibility than '`GDB_SHLIB_ROOT`' because it allows you to specify more than one path. However, there are some cases in which you may want to choose to use '`GDB_SHLIB_ROOT`'.

For example, if you have more than one shared library with the same name but different path names, you may want to use '`GDB_SHLIB_ROOT`' because GDB searches for libraries based on the full path name.

Note that '`GDB_SHLIB_PATH`' may not give you the results you expect because GDB searches for libraries that match only the name, regardless of the path, and always accepts the first library that matches the name.

For example, if you want to use '`/tmp/usr/lib/libsubs.sl`' and '`/tmp/usr/share/lib/libsubs.sl`', you can set '`GDB_SHLIB_ROOT`' to '`/tmp`'. Now whenever GDB encounters a library with the name '`/usr/lib/libsubs.sl`' and '`/usr/share/lib/libsubs.sl`', GDB looks at '`/tmp/usr/lib/libsubs.sl`' and '`/tmp/usr/share/lib/libsubs.sl`' respectively.

12.3 Errors reading symbol files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the `set complaints` command (see Section 17.6 [Optional warnings and messages], page 252).

The messages currently printed, and their meanings, include:

`inner block not inside outer block in` *`symbol`*

The symbol information shows where symbol scopes begin and end (such as at the start of a function or a block of statements). This error indicates that an inner scope block is not fully contained in its outer scope blocks.

GDB circumvents the problem by treating the inner block as if it had the same scope as the outer block. In the error message, *symbol* may be shown as "`(don't know)`" if the outer block is not a function.

`block at` *`address`* `out of order`
: The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so.

  GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine what source file is affected by specifying `set verbose on`. See Section 17.6 [Optional warnings and messages], page 252.)

`bad block start address patched`
: The symbol information for a symbol scope block has a start address smaller than the address of the preceding source line. This is known to occur in the SunOS 4.1.1 (and earlier) C compiler.

  GDB circumvents the problem by treating the symbol scope block as starting on the previous source line.

`bad string table offset in symbol` *`n`*
: Symbol number *n* contains a pointer into the string table which is larger than the size of the string table.

  GDB circumvents the problem by considering the symbol to have the name `foo`, which may cause other problems if many symbols end up with this name.

`unknown symbol type 0x`*`nn`*
: The symbol information contains new data types that GDB does not yet know how to read. `0x`*`nn`* is the symbol type of the uncomprehended information, in hexadecimal.

  GDB circumvents the error by ignoring this symbol information. This usually allows you to debug your program, though certain symbols are not accessible. If you encounter such a problem and feel like debugging it, you can debug `(gdb)` with itself, breakpoint on `complain`, then go up to the function `read_dbx_symtab` and examine `*bufp` to see the symbol.

`stub type has NULL name`
: GDB could not find the full definition for a struct or class.

`const/volatile indicator missing (ok if using g++ v1.x), got...`
: The symbol information for a C++ member function is missing some information that recent versions of the compiler should have output for it.

`info mismatch between compiler and debugger`
: GDB could not parse a type specification output by the compiler.

13 Specifying a Debugging Target

A *target* is the execution environment occupied by your program.

Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the `file` or `core` commands. For HP-UX specific information, see ⟨undefined⟩ [HP-UX Targets], page ⟨undefined⟩. When you need more flexibility—for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection you can use the `target` command to specify one of the target types configured for GDB (see Section 13.2 [Commands for managing targets], page 109).

13.1 Active targets

There are three classes of targets: processes, core files, and executable files. GDB can work concurrently on up to three active targets, one in each class. This allows you to (for example) start a process and inspect its activity without abandoning your work on a core file.

For example, if you execute '`gdb a.out`', then the executable file `a.out` is the only active target. If you designate a core file as well presumably from a prior run that crashed and coredumped, then GDB has two active targets and uses them in tandem, looking first in the corefile target, then in the executable file, to satisfy requests for memory addresses. (Typically, these two classes of target are complementary, since core files contain only the contents of the program read-write memory, variables, machine status etc. while the executable files contain only the program text and initialized data.)

When you type `run`, your executable file becomes an active process target as well. When a process target is active, all GDB commands requesting memory addresses refer to that target; addresses in an active core file or executable file target are obscured while the process target is active.

Use the `core-file` and `exec-file` commands to select a new core file or executable target (see Section 12.1 [Commands to specify files], page 103). To specify as a target a process that is already running, use the `attach` command (see Section 4.7 [Debugging an already-running process], page 27).

13.2 Commands for managing targets

`target` *`type parameters`*

Connects the GDB host environment to a target machine or process. A target is typically a protocol for talking to debugging facilities. You use the argument *type* to specify the type or protocol of the target machine.

Further *parameters* are interpreted by the target protocol, but typically include things like device names or host names to connect with, process numbers, and baud rates.

The `target` command does not repeat if you press ⟨RET⟩ again after executing the command.

`help target`
: Displays the names of all targets available. To display targets currently selected, use either `info target` or `info files` (see Section 12.1 [Commands to specify files], page 103).

`help target` *`name`*
: Describe a particular target, including any parameters necessary to select it.

`set gnutarget` *`args`*
: GDB uses its own library BFD to read your files. GDB knows whether it is reading an *executable*, a *core*, or a *.o* file; however, you can specify the file format with the `set gnutarget` command. Unlike most `target` commands, with `gnutarget` the `target` refers to a program, not a machine.

> *Warning:* To specify a file format with `set gnutarget`, you must know the actual BFD name.

See Section 12.1 [Commands to specify files], page 103.

`show gnutarget`
: Use the `show gnutarget` command to display what file format `gnutarget` is set to read. If you have not set `gnutarget`, GDB will determine the file format for each file automatically, and `show gnutarget` displays '`The current BDF target is "auto"`'.

Here are some common targets (available, or not, depending on the GDB configuration):

`target exec` *`program`*
: An executable file. '`target exec` *`program`*' is the same as '`exec-file` *`program`*'.

`target core` *`filename`*
: A core dump file. '`target core` *`filename`*' is the same as '`core-file` *`filename`*'.

`target remote` *`dev`*
: Remote serial target in GDB-specific protocol. The argument *dev* specifies what serial device to use for the connection (e.g. '`/dev/ttya`'). `target remote` supports the `load` command. This is only useful if you have some other way of getting the stub to the target system, and you can put it somewhere in memory where it won't get clobbered by the download.

`target sim`
: Builtin CPU simulator. GDB includes simulators for most architectures. In general,

```
target sim
load
run
```

works; however, you cannot assume that a specific memory map, device drivers, or even basic I/O is available, although some simulators do provide these.

Some configurations may include these targets as well:

`target nrom` *`dev`*
: NetROM ROM emulator. This target only supports downloading.

Different targets are available on different configurations of GDB; your configuration may have more or fewer targets.

Many remote targets require you to download the executable code once you have successfully established a connection.

`load` *`filename`*
: Depending on what remote debugging facilities are configured into GDB, the `load` command may be available. Where it exists, it is meant to make *filename* (an executable) available for debugging on the remote system—by downloading, or dynamic linking, for example. `load` also records the *filename* symbol table in GDB, like the `add-symbol-file` command.

  If your GDB does not have a `load` command, attempting to execute it gets the error message "`You can't do that when your target is ...`"

  The file is loaded at whatever address is specified in the executable. For some object file formats, you can specify the load address when you link the program; for other formats, like a.out, the object file format specifies a fixed address.

  `load` does not repeat if you press ⟨RET⟩ again after using it.

13.3 Choosing target byte order

Some types of processors, such as the MIPS, PowerPC, and Hitachi SH, offer the ability to run either big-endian or little-endian byte orders. Usually the executable or symbol will include a bit to designate the endian-ness, and you will not need to worry about which to use. However, you can adjust the processor byte order manually using one of the following commands:

`set endian big`
: Instruct GDB to assume the target is big-endian.

`set endian little`
: Instruct GDB to assume the target is little-endian.

`set endian auto`
: Instruct GDB to use the byte order associated with the executable.

`show endian`
: Display GDB's current idea of the target byte order.

Note that these commands merely adjust interpretation of symbolic data on the host, and that they have absolutely no effect on the target system.

14 HP-UX Configuration-Specific Information

While nearly all GDB commands are available for all native and cross versions of the debugger, there are some exceptions. This chapter describes features, commands and, enhancements available only on HP-UX.

14.1 Summary of HP Enhancements to GDB

WDB provides the following features in addition to the standard GDB features:

- Support for debugging memory problems.
- Support for heap usage reporting
- The `min-heap-size <num>` option for `set heap-check` command reports the heap allocations that exceed the specified number `<num>` of bytes based on the cumulative number of bytes that are allocated at each call-site, which is inclusive of multiple calls to `malloc` at a particular call site.
- Heap checking commands `info heap high-mem` and `set heap-check high-mem-count X_number` .
- Commands `which`, `exit`, `info heap process` and, `info heap arena`.
- WDB supports the `+check` compiler option on Integrity systems to invoke batch RTC, to determine runtime memory problems.
- Enhanced batch RTC support, for better reporting, and options have replaced old ones.
- RTC heap corruption checks for calls to `strcpy()`, `memset()`, and `memcpy()` have been added.
- Support for memory checking analysis for user defined memory management routines.
- High water mark records the number of times the `brk()` value changes.
- Heap analysis on programs with pending signals using the `info leak` command.
- Commands `info module ADDRESS`, `show envvars`, and `info corruption`.
- Command line option (`-pid` or `-p`) to attach to an existing process
- Support for debugging kernel threads and user threads.
- Support for enabling and disabling threads.
- Thread debugging commands `set thread-check on/off`, `info thread [thread-id]`, `info mutex [mutex- id]`, `info condvar [condvar-id]`, and, `info rwlock [rwlock-id]`.
- Enhanced thread debugging options for `set thread-check` command such as `recursive-relock`, `num-waiter`, `stack-util`, `thread-exit-no-join-detach`, `thread-exit-own-mutex`, `cv-wait-no-mx`, `cv-multiple-mxs`, `mixed-sched-policy`, and `unlock-not-own`
- Serial debugging of a parent and child process.
- Support for Parallel Processing limited to `pthread` parallelism, but not the compiler-generated parallelism, e.g. with directives.
- Implementation of `ask mode` for set `follow-fork-mode`.
- Support for setting breakpoints using shared library name.

- Support for core file commands `packcore`, `unpackcore`, `getcore`, `dumpcore` and `info rtti` *`address`*.
- On PA-RISC systems, sanity check for core files dumped by hardware generated signals can be performed. HP WDB can detect and warn the user about certain cases of corrupted core files.
- Inline support is on by default on Integrity systems. On PA-RISC, the inline support is still `off` by default.
- For PA 64-bit applications, WDB can step into shared library bind-on-reference calls. This support is available for PA 32-bit as well.
- Interception of synchronous signals used by `sigwait()`, `sigwaitinfo()` and `sigtimedwait()` functions. These signals are displayed by WDB just like asynchronous signals but are always passed to the debugger whether `nopass` is set or not.
- Support for debugging hardware watchpoints and shared libraries.
- For PA 64-bit applications, WDB can step into shared library bind-on-reference calls. This support is available for PA 32-bit as well.
- Implementation of `-mapshared` option to suppress mapping all the shared libraries in a process private.
- Support for deferred breakpoints on `dlopened` and `shl_loaded` libraries with stripped main program.
- Additional support for procedural breakpoints.
- C99 variable arrays implemented on Integrity systems. This support is available on PA-RISC as well.
- The `show envvars` command can be used to print information on environment variables supported by WDB.
- Support for handling `__fpreg` data type.
- Support for making changes to the program when debugging without having to re-compile or re-link it.
- Hardware breakpoints on Integrity systems.
- Support for debugging PA-RISC Applications on Integrity systems.
- Unwinding Java stack frames on Integrity systems. The 64-bit version of gdb can unwind through Java stack frames using the shared library in the Java product.The 64-bit library is part of the JDK 1.4.2.10 and JDK 1.5.0.03 products.
- Enhanced `nexti` and `stepi` commands. The WDB `nexti` and `stepi` commands, prints the assembly instruction along with the next source line.
- Enhanced `info symbol ADDRESS` command. The `info symbol ADDRESS` command has been enhanced to search for a symbol at the given address. Previously, the `info symbol` command could be used only to search the global namespace.
- Enhanced Java debugging support.
- Support for debugging C++ programs running under Linux Runtime Environment.
- Support for `stop in/at` dbx commands.
- Support for GNU GDB logging commands.

- Support for persistent display of expressions across re-runs. The user does not need to re-initiate the display settings for every run.
- Viewing wide character and wide-character strings of type `wchar_t` using the `type print` command.
- Support for debugging of executables with method and expressions involving covariant return types
- Support for commands, `catch throw`, `catch catch`, and `info catch`, for debugging exception handlers in C++ on Integrity systems, with the aCC compiler A.06.00 and later.
- Support for the `steplast` command for C and C++. However, `steplast` command is not supported.
- Support for dumping an array into an ASCII file.
- Support for the Fortran array slices.
- Source level debugging of Fortran applications that contain nested routines.
- Support for making command line calls in a stripped executable.
- Support for a terminal user interface that provides view of source while debugging at the WDB command line.
- Support for debugging 32-bit and 64-bit PA-RISC programs as well as 64-bit Itanium programs .
- Support for assembly-level debugging.
- Support for a subset of `xdb` commands, enabled with `-xdb` mode.
- Support for Java/C/aCC++ stack unwinding with Java SDK version 1.3.1.02 or later for HP-UX.
- Visual Interface for HP WDB `-tui` mode supports output logging
- Command line calls for 64-bit PA-RISC applications that are not linked with `end.o`.
- Command `watch_target`.
- Command line option `set display-full-path` that displays the full pathname of the source file name while printing the frame information.
- Command line option `set dereference [on |off]` when `off`, WDB does not dereference `char *` variables by default.
- Support for `show macro`, and `macro expand` to view, and expand macro expressions.
- Support for evaluating macros.
- Support for printing the execution path entries in the current frame, or thread.

  Note: For new commands see HP WDB Release Notes available at http://www.hp.com/go/wdb.

14.2 HP-UX dependencies

14.2.1 Linker Dependencies

Several features available in WDB depend on specific versions of the linker or the compiler.

- **Linker patch required for +objdebug**

  For releases prior to HP-UX 11i v2 (for IA) and HP-UX 11i v1 (for PA-RISC), you must install the latest linker patch to generate object modules that enable faster linking and smaller executable file sizes for large applications. See your your compiler release notes for more details.
- **Support for debugging incrementally linked 64-bit programs**

  This feature requires linker version B.11.18 or later on HP-UX 11i v1.
- **Support to automatically preload `librtc.sl` with `chatr +mem_check` option.** This feature requires linker version B.11.61 and later on HP 9000 systems, and linker version B.12.46 and later on Integrity systems.

14.2.2 Dependent Standard Library Routines for Run Time Checking

The Run Time Checking feature (Interactive and Batch Mode) of WDB cannot be used with applications that re-define or over-ride the default system-supplied versions of the standard library routines under `libc.so` and `libdld.so`. The following standard libraries are dependencies for runtime checking:

`abort()`	`atoi()`
`chdir()`	`dlsym()`
`strstr()`	`strcat()`
`ctime()`	`dlclose()`
`memchr()`	`strrchr()`
`dlgetname()`	`dlget()`
`clock_gettime()`	`strlen()`
`dlhook()`	`dlmodinfo()`
`environ()`	`getenv()`
`strlen()`	`execl()`
`exit()`	`fclose()`
`fork()`	`strdup()`
`fopen()`	`fprintf()`
`fscanf()`	`sscanf()`
`strcasecmp()`	`getcwd()`
`getpagesize()`	`getpid()`

`srand()`	`unlink()`
`uwx_register_callbacks()`	`lseek()`
`open()`	`sprintf()`
`strcmp()`	`printf()`
`pthread_self()`	`putenv()`
`shmctl()`	`strchr()`
`rand()`	`uwx_self_lookupip()`
`shl_get()`	`shl_unload()`
`shl_findsym ()`	`strtok_r()`
`time()`	`uwx_get_reg()`
`shl_get_r()`	`perror()`
`uwx_init()`	`uwx_self_copyin()`
`creat()`	`uwx_step()`
`write()`	`uwx_self_init_context()`
`pthread_getschedparam()`	`uwx_self_init_info()`
`uwx_register_alloc_cb()`	`U_STACK_TRACE()`
`close()`	`strchr()`

The runtime checking (of dynamic memory, libraries, and pthreads) in the debugger relies on the semantic and standard behavior of these library routines. Run Time Checking results in unexpected and unpredictable behavior when used with applications that substitute or re-define these library routines.

Before enabling the Run Time Checking feature in WDB, use the `nm(1)` command to determine if the application or the dependent libraries in the application re-define or substitute these standard library routines

14.3 Supported Platforms and Modes

- **Supported Platforms**

  HP WDB supports source-level debugging of programs written in HP C, HP aC++, and Fortran 90 on Integrity systems running on HP-UX 11i v2 or later and PA- RISC systems running HP-UX 11i v1 and later.
- **Support for assembly-level debugging**

  HP WDB provides support for assembly-level debugging.
- **Support for automatic loading of debug information**

  Debug information is automatically loaded from modules when an application is compiled with the `+objdebug` option.
- **Support for debugging PA-RISC programs on Itanium-based systems**

  You can debug PA-RISC applications and core files on Itanium-based systems. When you start HP WDB, if the debug target is a PA-RISC binary program, the debugger automatically loads PA-RISC WDB. The PA-RISC version of HP WDB is provided as part of the HP-UX operating system.

- **Support for debugging large core files (> 2GB)**

  HP WDB supports debugging of core files with sizes more than 2 GB.
- **Support co-variant type**

  HP WDB can step into a co-variant function. The compiler-generated function called *thunks*, which is used internally by the compiler to support co-variant return type, is not shown when you do a backtrace or switch from one frame to another frame. Similarly, using a `finish` or `return` command at a co-variant callee function directly returns the control back to the caller of thunks.
- **New attach command line options and handling (-pid or -p)**

  HP WDB accepts `-pid` or `-p` followed by a process ID to attach a running process to the debugger.

  Note:

  HP WDB cannot be attached to a process that is traced by tools which use `ttrace`, such as Caliper, adb, and tusc. The debugger displays the following error message on attempting to attach to such a process:

```
Attaching to process <pid> failed.
Hint: Check if this process is already being traced by another gdb or
      other ttrace tools like caliper and tusc.
Hint: Check whether program is on an NFS-mounted file-system.
      If so, you will need to mount the file system with the "nointr" option
with mount(1) or make a local copy of the program to resolve this problem.
```

14.4 HP-UX targets

On HP-UX systems, GDB has been configured to support debugging of processes running on the PA-RISC and Itanium architectures. This means that the only possible targets are:

- An executable that has been compiled and linked to run on HP-UX. This includes binaries that have been marked as `SHMEM_MAGIC`.
- A live HP-UX process, either started by WDB (with the `run` command) or started outside of WDB and attached to (with the `attach` command).
- A core file generated by an HP-UX process that previously aborted execution.

GDB on HP-UX has not been configured to support remote debugging, or to support programs running on other platforms.

WDB can only debug C++ programs compiled with HP aC++, the ANSI-compatible C++ compiler.

14.5 Support for Alternate root

HP WDB supports alternate root functionality, which is helpful when you do not want to use the system-installed HP WDB or its components.

The environment variable `WDB_ROOT` specifies the alternate root for HP WDB. You must specify a structure similar to the default `/opt/langtools` used for HP WDB. You can use the environment variable `GDB_ROOT` to specify an alternate root for GDB.

If you specify both `WDB_ROOT` and `GDB_ROOT`, the value for `GDB_ROOT` is ignored.

HP WDB supports these environment variables to override the location of different component of HP WDB.

Defined Variable	WDB Location	GDB location	librtc.sl location
None	/opt/langtools/bin	/opt/langtools/bin	/opt/langtools/lib
WDB_ROOT	$WDB_ROOT/bin	$WDB_ROOT/bin	$WDB_ROOT/lib
GDB_ROOT	n/a	$GDB_ROOT/bin	/opt/langtools/bin
GDB_SERVER	n/a	$GDB_SERVER	n/a
LIBRTC_SERVER	n/a	n/a	$LIBRTC_SERVER

Note: If you define `WDB_ROOT` or `GDB_ROOT` but do not create the correct directory structure below it, the debugger may fail.

14.6 Specifying object file directories

GDB enables automatic loading of debug information from object modules when an application is compiled with the `+objdebug` option.

GDB uses the full path name to the object module files and searches the same directories for source files.

Although this behavior is transparent, you can control when and how object files are loaded with three commands:

`objectdir` *`path`*
: Specifies a colon (:) separated list of directories in which GDB searches for object files. These directories are added to the beginning of the existing `objectdir` path. If you specify a directory that is already in the `objectdir` path, the specified directory is moved up in the `objectdir` path so that it is searched earlier.

  GDB recognizes two special directory names: `$cdir`, which refers to the compilation directory (if available) and `$cwd`, which tracks GDB's current working directory.

`objectload` *`file.c`*
: Causes GDB to load the debug information for *file.c* immediately. The default is to load debug information from object modules on demand.

`objectretry` *`file.c`*
: Forces GDB to retry loading an object file if GDB encounters a file error while reading an object module. File errors that might cause this include incorrect permissions, file not found, or if the `objectdir` path changes. By default, GDB does not try to read an object file after an error.

`pathmap`
: Enables you to define a list of substitution rules to be applied to path names to identify object files and the corresponding source files. The `pathmap` command, however, may not find source files if the object files are not available.

  This minimizes or eliminates the need to specify multiple `objectdir` commands when object files are moved from the compilation directories or when compilation directories are mounted over NFS.

To use this feature, the program must be compiled with the `+objdebug` option. For information on how `pathmap` works type help pathmap at the HP WDB prompt.

If the debugger cannot find the source: files.

1. Make certain the files were compiled with the `'-g'` switch. Type *`info sources`* to find the list of files that the debugger knows were compiled with `'-g'`.
2.

   Make certain that the debugger can find the source file. Type *`show dir`* to find the list of directories the debugger uses to search for source files and type *`set dir`* to change that path.

   On HP-UX, the debug information does not contain the full path name to the source file, only the relative path name that was recorded at compile time. Consequently, you may need several `dir` commands for a complex application with multiple source directories. One way to do this is to place them in a '`.gdbinit`' file placed in the directory used to debug the application.

   A sample of the '`.gdbinit`' file might look like the following:

   ```
   dir /home/fred/appx/system
   dir /home/fred/appx/display
   dir /home/fred/appx/actor
   dir /home/fred/appx/actor/sys
        ...
   ```

   When you compile the program with the `+objdebug` option, the debugger may find the source files without using the `dir` command. This happens because the debugger stores the full path name to the object files and searches for source files in the same directories.

14.7 Fix and continue debugging

Fix and continue enables you to see the result of changes you make to a program you are debugging without having to re-compile and re-link the entire program.

For example, you can edit a function and use the `fix` command, which automatically re-compiles the code, links it into a shared library, and continues execution of the program, without leaving the debugger.

With Fix and Continue, you can experiment with various ways of fixing problems until you are satisfied with the correction, before you exit the debugger.

The advantages include:

- You do not have to recompile and relink the entire program.
- You do not have to reload the program into the debugger.
- You can resume execution of the program from the fix location.
- You can speed up the development cycle.

Note:

> Fix and Continue is only supported with the most recent versions of HP C and HP aC++ on PA-RISC systems.

In command-line mode, you use the `edit` command before invoking the `fix` command.

The `edit` command has the following syntax:

`edit` *file1 file2*

where *file*

represents one or more source files for the current executable. If you do not specify a file name, WDB edits the currently open source file.

When you edit a file with the `edit` command and save the changes, the original source file contains the changes, even if you do not use the `fix` command to recompile the program in the debugger.

14.7.1 Fix and Continue compiler dependencies

Fix and Continue is supported only for PA-RISC on HP-UX 11.x with these compilers:

- HP C/ANSI C A.11.01.20, or later
- HP aC++ A.03.25, or later
- HP Fortran 90 2.4, or later

14.7.2 Fix and Continue restrictions

Fix and Continue has the following restrictions and behaviors:

- You cannot recompile code that has been optimized.
- You cannot add, delete, or reorder the local variables and parameters in a function currently active on the stack.
- If you fix a routine in a file that contains function pointers, those function pointers become invalid and will likely cause the program to receive a `SIGSEGV` error if the pointers are used.
- You cannot change the type of a local variable, file static, global variable, or parameter of a function.
- You cannot add any function calls that increase the size of the parameter area.
- You cannot change a local or file static or global variable to be a register variable, and vice-versa.
- You cannot add an `alloca()` function to a frame that did not previously use `alloca()`.
- New structure fields can be added at the end of a structure object, not in the middle of a structure. New fields are only accessible by the modified files. Old structure fields remain intact; no swapping of them is permitted.
- If the redefined function is in the call stack but not on the top of the call stack, the modified code will not be executed when the execution resumes.

  The modified function will be executed when it is called next time, or a rerun.
- Breakpoints in the original source file are moved to the modified file. Breakpoints in the already instantiated functions on the call stack in the original file are lost.

- If you change the name of a function and there was a breakpoint set to the old function, WDB does not move the breakpoint to the new function. The old breakpoint is still valid.
- If the number of lines of the modified file is different from that of the original file, the placement of breakpoints may not be correct.
- When the program resumes, the program counter is moved to the beginning of the same line in the modified function. The program counter may be at the wrong line.

14.7.3 Using Fix and Continue

When WDB recompiles a fixed source file, it uses the same compiler and the same options that were used to create the original executable. If the compiler generates any syntax errors or it encounters any of the restrictions, WDB does not patch the changes into the executable image being debugged.

After you successfully recompile the changes, WDB uses the fixed version of the code when you use any of the execution commands such as `step`, `run`, or `continue`.

When you use the `edit` command, WDB then monitors any edited source files for additional changes. After you enter the initial `fix` command, WDB checks for additional saved changes to a source file each time you enter a program execution command. If a saved source file has been changed, WDB asks if you want to fix the changed source, allowing you to apply repeated fixes without explicitly entering the `fix` command.

The Fix and Continue facility enables you to make the following changes:

- Change existing function definitions.
- Disable, reenable, save, and delete redefinitions
- Adding global and file static variables.
- Add new structure fields to the end of a structure type object.
- Set breakpoints in and single-step within redefined code.

Note:

> You must rebuild the program after you use the `fix` command because the changes you make are temporarily patched into the executable image. The changes are lost if you load a different executable and are not reflected in the original executable when you exit the debugger.

14.7.4 Example Fix and Continue session

This example shows how you can make and test changes to a function without leaving the debugger session.

Here is a short sample C program with an error:

```
int sum (num)   int num;
{
  int j, total = 0;
  for (j = 0; j <= num; j++)
    total += num;
}

main()
{
   int num = 10;
   printf("The sum from 1 to %d is = %d\n", num, sum(num));
}
```

1. Compile the program.

   ```
   cc sum.c -g -o mysum

   /usr/ccs/bin/ld: (Warning) At least one PA 2.0 object file
   (sum.o) was detected.
    The linked output may not run on a PA 1.x system.
   ```

2. Run the program.

   ```
   ./mysum

   The sum from 1 to 10 is = 0
   ```

   This result is obviously wrong. We need to debug the program.

3. Run the debugger:

   ```
   gdb mysum

   HP gdb 3.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
   Copyright 1986 - 2001 Free Software Foundation, Inc.
   Hewlett-Packard Wildebeest 3.0 (based on GDB ) is covered by the
   GNU General Public License. Type "show copying" to see the
   conditions to change it and/or distribute copies. Type
   "show warranty" for warranty/support.
   ..
   ```

   If the `TERM` environment variable is not set to `hpterm`, start the debugger and set the terminal type for editing in WDB with this command (`ksh` shell):

   ```
   TERM=hpterm gdb mysum
   ```

   The problem might be that there is no `return` for the `num` function. You can correct this without leaving the debugger.

4. Set a break point at `main`:

   ```
   (gdb) b main
   Breakpoint 1 at 0x23f8: file sum.c, line 11.
   ```

5. Run the program:

   ```
   (gdb) run
   Starting program: /tmp/hmc/mysum

   Breakpoint 1, main () at sum.c:11
   11          int num = 10;
   ```

6. When the program stops at the break point, use the `edit` command to make changes to the source file.

Because you are going to edit the current file, you do not need to specify a source file name.

```
(gdb) edit
```

The `edit` command opens a new terminal session using your environment variable settings for terminal and editor. The debugger automatically loads the source file.

7. Make the necessary changes. In this case, add:

   ```
   return total;
   ```

   to the function named `num`.
8. Save the edited source file and exit the editor. This saves the changes in the actual source file for the program.
9. Use the `fix` command to recompile the program to see the results of the changes:

   ```
   (gdb) fix
   Compiling /dev/src/sum.c...
   Linking...
   Applying code changes to sum.c.
   Fix succeeded.
   ```

   The `fix` command creates a new executable that includes the changes you made to the source file.

   The debugger automatically uses the new executable and picks up the debugging session where you stopped before using the `edit` command.

   For example, you can continue stepping through the program and you will find the new `return total;` statement in the source view. You can print the value of `total`, and see that the result is 110.
10. When you finish with the debugging session, you can exit the debugger normally:

    ```
    (gdb) q
    The following modules in /dev/src/mysum have been fixed:
    /dev/src/sum.c
    Remember to remake the program.
    ```

    The debugger message lists the source files that you have changed during the debugging session.

    Note:

    You must rebuild the program after you use the `fix` command because the changes you make are temporarily patched into the executable image. The changes are lost when you exit the debugger or you load a different executable.

14.8 Inline Support

HP WDB enables you to debug inline functions in applications compiled with `-g` option. To enable inline debugging in HP 9000 systems, the applications must be compiled with the `+inline_debug` option (introduced in the A.03.65 and later versions of the HP aC++ compiler). In Integrity systems, the applications that are compiled with `-g` option support inline debugging by default and require no additional options. Compiler versions A.06.02

and later support the inline debugging feature in Integrity systems.

14.8.1 Inline Debugging in HP 9000 Systems

To debug inline functions in HP 9000 systems, complete the following steps:

Step 1:

Compile the source files with the `+inline_debug` option.
For example:

```
/opt/aCC/bin/aCC -g +inline_debug test.c
```

Step 2:

Inline debugging is enabled by default. To explicitly enable or disable inline debugging, complete either of the following steps before loading the application to the debugger:

```
$ gdb --inline=<on/off> a.out
```

or

```
(gdb) set inline-debug <on/off>
```

Step 3:

You can use the following commands for debugging inline functions in HP 9000 systems:

```
step
next
list
backtrace
frame <n>
info locals
info args
```

The following commands are not available for debugging inline functions in HP 9000 systems:

```
breakpoint
info frame
disassembly
```

Note:

Inline debugging commands are not available for inlined template functions and inlined functions which are defined after the call site.

14.8.2 Inline Debugging in Integrity Systems

In Integrity systems, applications that are compiled with `-g` option support inline debugging by default. Compiler versions A.06.02 and later support the inline debugging feature in Integrity systems and require no additional options.

WDB 5.6 and later versions enable you to set and modify breakpoints in inline functions for programs compiled with optimization level less than +O2. The breakpoint features for inline functions are introduced as additional options in the `set inline-debug` command.

You can toggle the options for inline debugging by entering either of the following commands:

```
(gdb) set inline-debug  <option>
```

or

```
$ gdb --inline= <option>
```

The following options available for the `set inline-debug` command:

- `on`
- `off`
- `inline_bp_all`
- `inline_bp_individual`

The `set inline-debug on` command enables the inline debugging feature without the inline breakpoints options in Integrity systems. This command is enabled by default.

The `set inline-debug off` command disables the inline debugging feature. You can disable inline debugging by entering this command before attaching the debugger to the application.

The `set inline-debug inline_bp_all` command enables you to set and modify breakpoints on all instances of a particular inline function. It also enables the inline debugging feature. A single instance of the specified inline function is displayed as a representative instance for all the instances of the specified inline function. This creates a single-breakpoint illusion for multiple instances of the inline function. You can set and modify breakpoints on all the instances of the inline functions by setting and modifying breakpoints on the displayed instance of the inline function. You must enter this command before attaching the debugger to the application.

The `set inline-debug inline_bp_individual` command enables you to set and modify breakpoints on a specific instance of an inline function. It also enables the inline debugging feature. All instances of the inline function are displayed separately with individual breakpoint occurrences. You can set or delete individual breakpoints on a specific instance of an inline function without modifying the breakpoints on other instances of the inline function. You must enter this command before attaching the debugger to the application.

Limitations:

- The inline breakpoint features are not available for programs that are compiled with +O2 optimization level and above.

- The inline breakpoint features can degrade performance of the application that is being debugged. You can explicitly disable the breakpoint features when the features are not required and continue to use other inline debugging features, such as `step` and `next`.

14.8.2.1 Debugging Inline Functions in Integrity Systems

To debug inline functions in Integrity systems, complete the following steps:

Step 1:

The application must be compiled with the `-g` option for inline debugging. No additional options are required.
For example:

```
/opt/aCC/bin/aCC -g test.c
```

Step 2:

Inline debugging without the breakpoint feature is enabled by default. You can modify the inline debugging settings by toggling the options for the `set inline-debug` command.

- To enable inline debugging without inline breakpoint support, enter either of the following commands:

  ```
  (gdb) set inline-debug on
            or
  $ gdb --inline = on
  ```

- To set and modify breakpoints collectively on all instances of inline functions and enable inline debugging, enter either of the following commands:

  ```
  (gdb) set inline-debug inline_bp_all
            or
  $ gdb --inline = inline_bp_all
  ```

- To set and modify individual breakpoints on specific instances of inline functions and enable inline debugging, enter either of the following commands before debugging the application:

```
(gdb) set inline-debug inline_bp_individual
        or
$ gdb --inline = inline_bp_individual
```

- To disable inline debugging, enter either of the following commands before debugging the application:

```
(gdb) set inline-debug off
        or
$ gdb --inline= off
```

Step 3:

You can use the following commands for debugging inline functions in Integrity systems:

```
step
next
list
backtrace
frame <n>
info locals
info args
breakpoint
```

The following commands are not available for debugging inline functions in Integrity systems:

```
info frame
disassembly
```

14.9 Debugging Macros

HP WDB 5.7 and later versions of the debugger enable you to display and evaluate macro definitions for programs running on Integrity systems. This feature is available only for compiler versions A.06.15 and later.

14.9.1 Viewing and Evaluating Macro Definitions

HP WDB 5.7 and later versions of the debugger provide the following support for debugging macros:

- Displaying Macro Definitions

  HP WDB provides the following commands to display macro definitions:

  - `show macro [macro-name]` or `info macro [macro-name]`

    Displays the macro definition, source file name, and the line number. For example:

```
(gdb) info macro VAR2
Defined at scope.c:21
#define VAR2 201
```

- `macro expand [macro-name]`

  Expands the macro and the parameters in the macro. If there are any parameters in the macro, they are substituted in the macro definition when the definition is displayed.

  For example:

  ```
  #define YY 6
  #define MAC (67 + YY)
  ...
  $ gdb
  ...
  (gdb) macro expand MAC
  expands to: (67 + 6)
  ```

- Evaluating Macros

  HP WDB enables you to evaluate a macro and display the output. You can evaluate the macro by using the commonly used gdb commands for evaluating and displaying expressions, such as `print`. HP WDB supports the evaluation of macros with variables, constants, complex algebraic expressions involving variables, nested macros, and function calls. HP WDB does not support the evaluation of macros with multiple statements in the macro definitions, or the evaluation of macros with stringifying and pasting tokens in the macro definitions.

14.9.1.1 Compiler Options to Enable Macro Debugging

To enable macro debugging, the program must be compiled with the

`+macro_debug=[all|none|ref]` compiler option.

Additionally, the program must be compiled with one of the `-g` options (`-g`, `-g0`, or `-g1`) to enable macro debugging. For example:

```
cc -g +macro_debug=all -o sample sample.c
```

The following options are available for the `+macro_debug` compiler option:

`all` To view and evaluate all the macro expressions in the program, you must compile the program with `+macro_debug=all`. This option can cause a significant increase in object file size.

`ref` To view and evaluate only the reference macros in the program, you must compile the program with `+macro_debug=ref`. This is the default for `-g`, `-g0`, or `-g1`.

`none` To disable macro debugging, you must compile the program with `+macro_debug=none`

The macro debugging features are supported for `+objdebug` and `+noobjdebug` compiler options.

14.9.2 Examples for Macro Debugging

The following example illustrates the use of the macro debugging:

Sample Program:

```
$ cat scope.c

#include <stdio.h>

#define USED1 100
#define USED2 200
#define UNUSED1 0
#define UNUSED2 0
#define DUMMY1
#define DUMMY2

int
main ()
{

  int val = USED1;

#undef UNUSED1
#undef USED2
#undef USED1
#define USED1 101
#define USED2 201

  val = USED1 + USED2;

#undef USED1
#undef UNUSED2
#undef USED2
#define USED1 102

  val = USED1;

  return 0;
}
```

Sample Debugging Session

- The following debugging session illustrates macro debugging when the program is compiled with `+macro_debug=all` option:

```
$ cc -g +macro_debug=all -o sc scope.c
$ gdb sc
HP gdb for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
```

```
...
(gdb) b 13
Breakpoint 1 at 0x40007d0:0: file scope.c, line 13 from sc.
(gdb) b 23
Breakpoint 2 at 0x40007d0:2: file scope.c, line 23 from sc.
(gdb) b 30
Breakpoint 3 at 0x40007e0:0: file scope.c, line 30 from sc.
(gdb) r
Starting program: sc

Breakpoint 1, main () at scope.c:13
13 {
(gdb) print USED1
100
(gdb) print USED1+10
110
(gdb) info macro USED1
Defined at scope.c:4
#define USED1 100
(gdb) info macro USED2
Defined at scope.c:5
#define USED2 200
(gdb) c
Continuing.
Breakpoint 2, main () at scope.c:23
23        val = USED1 + USED2;
(gdb) info macro USED1
Defined at scope.c:20
#define USED1 101
(gdb) info macro USED2
Defined at scope.c:21
#define USED2 201
(gdb) c
Continuing.

Breakpoint 3, main () at scope.c:30
30        val = USED1;
(gdb) info macro USED1
Defined at scope.c:28
#define USED1 102
(gdb) info macro USED2
The macro ‘USED2’ has no definition in the current scope.
(gdb)
```

- The following debugging session illustrates macro debugging when the program is compiled with `+macro_debug=referenced` option:

```
$ cc -g +macro_debug=referenced -o sc1 scope.c
$ gdb sc1
HP gdb for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
...
(gdb) b 13
Breakpoint 1 at 0x40007d0:0: file scope.c, line 13 from sc1.
(gdb) b 23
Breakpoint 2 at 0x40007d0:2: file scope.c, line 23 from sc1.
(gdb) b 30
Breakpoint 3 at 0x40007e0:0: file scope.c, line 30 from sc1.
(gdb) r
Starting program: sc1

Breakpoint 1, main () at scope.c:13
13 {
(gdb) print USED1
100
(gdb) print USED1+10
110
(gdb) info macro USED1
Defined at scope.c:4
#define USED1 100
(gdb) info macro USED2
The macro ‘USED2’ has no definition in the current scope.
(gdb) c
Continuing.

Breakpoint 2, main () at scope.c:23
23        val = USED1 + USED2;
(gdb) info macro USED1
Defined at scope.c:20
#define USED1 101
(gdb) info macro USED2
Defined at scope.c:21
#define USED2 201
(gdb) c
Continuing.

Breakpoint 3, main () at scope.c:30
30        val = USED1;
(gdb) info macro USED1
Defined at scope.c:28
#define USED1 102
(gdb) info macro USED2
The macro ‘USED2’ has no definition in the current scope.
(gdb) q
```

```
The program is running.  Exit anyway? (y or n) y.
```

14.10 Debugging Memory Problems

You can use WDB to find leaks, profile heap usage and detect other heap-related errors in HP C, HP aC++, and HP Fortran programs written for HP-UX 11.x systems. (Both 32-bit and 64-bit programs are supported.)

On HP-UX 11.x, the memory debugging features of WDB work with both single-threaded and multi-threaded programs that use POSIX threads.

For more information on memory debugging with WDB, see the *Debugging Dynamic Memory Usage Errors Using HP WDB* whitepaper at the HP WDB Documentation webpage at:

`http://www.hp.com/go/wdb`.

14.10.1 When to suspect a memory leak

You should suspect a memory leak in the code when you notice that the system is running out of swap space or running slower, or both.

Applications or non-kernel code (including daemons) that have memory leaks can eventually use up all swap space. You can run `top(1)` to verify whether the process data space (`SIZE`, `RES`) is growing more than you expect.

If the system is running out of swap space, programs will fail with out-of-memory (`ENOMEM`) errors or `SIGBUS` signals. In addition, the system might run slower and slower until it comes to a stop; all processes requiring swap to continue running will wait for it indefinitely.GDB allows you to catch out-of-memory conditions through runtime memory checking. Use the command `catch nomem` to detect out-of-memory conditions. GDB will stop whenever malloc returns `NULL` and allows you to look at the current context.

14.10.2 Memory debugging restrictions

Programs with these attributes are not supported:

- CMA or DCE threaded programs on 11.x (32-bit and 64-bit)
- Memory checking features. These features work only in programs that directly or indirectly call `malloc`, `realloc`, `free`, `mmap`, or `munmap` from the standard C library '`libc.sl`'.
- Programs that link the archive version of the standard C library, `libc.a`, or the core library, `libcl.a`, on HP-UX 11.x

  Note: Linker with version number B.11.19 or higher is required for debugging memory problems.
- From HP WDB 5.7 onwards, the archive version of the run time check library, `librtc.a`, is not available. You must use the shared version of the library, `librtc.[sl|so]`, instead.

14.10.3 Memory Debugging Methodologies

WDB enables you to debug memory problems in applications written for HP-UX 11.x or later using C, aC++, FORTRAN 77, and Fortran 90.

WDB provides several commands that help expose memory-related problems.

HP WDB offers the following memory-debugging capabilities:

- Reports memory leaks
- Reports heap allocation profile
- Stops program execution if bad writes occur with string operations such as strcpy and memcpy
- Stops program execution when freeing un-allocated or de-allocated blocks
- Stops program execution when freeing a block if bad writes occur outside block boundary
- Stops program execution conditionally based on whether a specified block address is allocated or de-allocated
- Scrambles previous memory contents at `malloc()` and `free()` calls
- Simulates and detects out-of-memory event errors
- Detects dangling pointers and dangling blocks
- Detects **in-block** corruption of freed blocks
- Specifies the amount of guard bytes for every block of allocated memory
- Displays the run time type information for C++ polymorphic objects

You can use any of the following methods to identify memory problems:

- See Section 14.10.4 [Debugging Memory in Interactive Mode], page 134
- See Section 14.10.5 [Debugging Memory in Batch Mode], page 138
- See Section 14.10.6 [Debugging Memory Interactively After Attaching to a Running Process], page 144

14.10.4 Debugging Memory in Interactive Mode

This section describes the various commands which help in debugging memory problems when the debugger is used in the interactive mode.

14.10.4.1 Commands for interactive memory debugging

To debug memory problems, use these commands:

`set heap-check [on|off]`
: This toggles the capability for detection of leaks, heap profiles, bounds checking, and checking for double free.

`info heap`
: Displays a heap report, listing information such as the start of heap, end of heap, heap size, heap allocations, size of blocks, and number of instances. The report shows heap usage at the point you use the `info heap` command. The

report does not show allocations that have already been freed. For example, if you make several allocations, free them all, and then you use `info heap`, the result does not show any allocations.

`info heap` *`filename`*
: Writes heap report output to the specified file.

`info heap` *`idnumber`*
: Produces detailed information on the specified heap allocation including the allocation call stack.

`show heap-check`
: Displays all current settings for memory checking.

`set heap-check interval < nn >`
: This command starts incremental heap growth profile. All allocations prior to the execution of this command are ignored from report. If incremental heap growth profile is already on, executing this command will reset the counters and start a fresh collection.Interval is specified in seconds.

`set heap-check repeat < nn >`
: This allows user to specify the number of intervals GDB should collect the incremental heap growth. The default value is 100. Every repeat of the interval tracks heap allocation during that interval.

  Example:

  `< gdb > set heap-check interval 10`

  `< gdb > set heap-check repeat 100`

  Here WDB will create 100 incremental apart heap profiles which are 10 seconds apart.

`set heap-check reset`
: GDB stores incremental heap growth data in an internal file. During one session data is appended to this file. If the session is very long, it is possible that this file may become very large. This command discards the data existing in the file and creates a new data file. Once this command is executed, the user cannot see the old data.

`info heap-interval < file name >`
: This command creates the report of heap growth. The data for each interval has the start and end time of the interval. If file name is mentioned a detailed report is written in the file.

`set heap-check leaks [on | off]`
: Controls WDB memory leak checking.

`info leaks`
: Displays a leak report, listing information such as the leaks, size of blocks, and number of instances.

`info leaks` *`filename`*
: Writes the complete leak report output to the specified file.

`info leak` *`leaknumber`*
: Produces detailed information on the specified leak including the allocation call stack.

`set heap-check block-size` *`num-bytes`*
: Instructs WDB to stop the program whenever it tries to allocate a block larger than *num-bytes* in size.

`set heap-check heap-size` *`num-size`*
: Instructs WDB to stop the program whenever it tries to increase the program heap by at least *num-bytes*.

`min-heap-size`
: This option reports the heap allocations that exceed the specified number `<num>` of bytes based on the cumulative number of bytes that are allocated at each call-site, which is inclusive of multiple calls to `malloc` at a particular call site.

`set heap-check watch` *`address`*
: Stops the program whenever a block at a specified address is allocated or deallocated.

`set heap-check null-check [N | random]`
: Force `malloc` to return `NULL` after `N` or random number of invocations of `malloc`. Use `set heap-check random-range N` and set `heap-check seed-value N` to define the seed value and range for random number calculation. Useful for simulating out-of-memory situations.

`set heap-check null-check-size N`
: Force `malloc` to return `NULL` after `N` bytes have been allocated by the program.

`catch nomem`
: Allows the user to get control of an out-of-memory event. The user can step through once the `nomem` event is caught to see whether the out-of-memory event is handled correctly.

`set heap-check free [on | off]`
: When set to on, forces WDB to stop the program when it detects a call to `free()` with an improper argument, a call to `realloc()` that does not point to a valid currently allocated heap block, or a call to free a block of memory, which has been corrupted. Refer to 'set heap-check bounds' for details on detecting memory corruption.

`set heap-check string [on | off]`
: Toggles validation of calls to `strcpy`, `strncpy`, `memcpy`, `memccpy`, `memset`, `memmove`, `bzero`, and, `bcopy`. WDB 5.6 and later versions of the debugger also validates calls to `strcat` and `strncat`.

`set heap-check bounds [on | off]`
: Allocates extra space at the beginning and end of a heap block during allocation and fills it with a specific pattern. When blocks are freed, WDB verifies whether these patterns are intact. If they are corrupted, an underflow or overflow must have occurred and WDB reports the problem. This option increases the program's memory requirements.

`set heap-check scramble [on | off]`
: Scrambles a memory block and overwrites it with a specific pattern when it is allocated or deallocated. This change to the memory contents increases the chance that erroneous behaviors, such as attempting to access space that is freed or depending on initial values of `malloc()` blocks, cause the program to fail.

`info dangling`
: Displays a list of all the dangling pointers and dangling blocks that are potential sources of memory corruption(may have false positives).

`info corruption`
: Checks for corruption in the currently allocated heap blocks.In addition, it lists the potential **in-block** corruptions in all the freed blocks.

`set heap-check min-leak-size` *`num`*
: Collects a stack trace only when the size of the leak exceeds the number of bytes you specify for this value. Larger values improve run-time performance. The default value is zero (0) bytes.

`set heap-check frame-count` *`num`*
: Controls the depth of the call stack collected. Larger values increase run time. The default value is four (4) stack frames.

`set heap-check header-size` *`num of bytes`*
: Sets the Header guard for each block of the allocated memory. The default number of bytes for the footer is 16 bytes if this option is not used. If the user specifies a value less than 16 for the number of bytes, the debugger ignores it and takes the default of 16 bytes. If the user specifies more than 16 bytes, then the debugger considers the largest and closest 16 byte integral from the user-specified value.

  Example:

  If the user specifies 60 bytes, the debugger takes it as 48 bytes. If the user specifies 65, the debugger considers 64 bytes.

`set heap-check footer-size` *`num of bytes`*
: Sets the **Footer** guard for each block of the allocated memory. The default number of bytes for the footer is one byte if this option is not used.

14.10.4.2 Example for interactive debugging session

This example describes checking a program running on HP-UX 11.x using linker version B.11.19 or later:

1. Link the program with `/usr/lib/libc.sl` instead of `libc.a`.
2. Run the debugger and load the program:

   ```
   > gdb a.out
   ```

3. Turn on leak checking:

   ```
   (gdb) set heap-check leaks on
   ```

4. Set one or more breakpoints in the code where you want to examine cumulative leaks:
   ```
   (gdb) b myfunction
   ```
5. Run the program in the debugger:
   ```
   (gdb) run
   ```
6. Use the `info` command to show list of memory leaks:
   ```
   (gdb) info leaks

   Scanning for memory leaks...done

   2439 bytes leaked in 25 blocks

   No.   Total bytes     Blocks     Address     Function
   0        1234           1       0x40419710   foo()
   1        333            1       0x40410bf8   main()
   2        245            8       0x40410838   strdup()

   [...]
   ```
   The debugger assigns each leak a numeric identifier.
7. To display a stack trace for a specific leak, use the `info leak` command and specify the number from the list associated with a leak:
   ```
   (gdb) info leak 2

   245 bytes leaked in 8 blocks (10.05% of all bytes leaked)
   These range in size from 26 to 36 bytes and are allocated
    in strdup ()
    in link_the_list () at test.c:55
    in main () at test.c:13
    in _start ()
   ```

14.10.5 Debugging Memory in Batch Mode

HP WDB supports batch mode memory leak detection, also called batch Run Time Checking (Batch RTC). Most of the memory debugging features supported in interactive mode are also supported in batch mode.

> **Note**:
>
> The batch mode commands may not always work when invoked through a shell script.

14.10.5.1 Setting Configuration Options for Batch Mode

You can specify the batch mode configuration through a configuration file called `rtcconfig`. The configuration file supports these variables:

`check_free=on|off (or) set heap-check free <on/off>`
: Enables detection of multiple incorrect free of memory

`check_heap|heap=on|off (or) set heap-check <on/off>`
: Enables heap profiling.

`check_leaks|leaks=on|off (or) set heap-check leaks <on/off>`
: Enables leak detection.

`check_string=on|off (or) set heap-check string <on/off>`
: Enables detection for writing out of boundary for `strcpy`, `strncpy`, `memcpy`, `memccpy`, `memset`, `memmove`, `bzero`, `bcopy`.

`check_bounds|bounds=on|off (or) set heap-check bounds <on/off>`
: Enables checking of bounds corruption.

`files=<file1:file2:...|fileN>`
: Specifies the executables for which memory leak detection is enabled. if files option is not specified, after setting `BATCH_RTC=on`, RTC will instrument ALL executables.

`frame_count=no_frames (or) set heap-check frame-count <no_frames>`
: Sets the number of frames to be printed for leak context.

`min_heap_size=block_size (or) set heap-check min-heap-size <block_size>`
: Sets the minimum block size to use for heap reporting.

`min_leak_size=block_size (or) set heap-check min-leak-size <block_size>`
: Sets the minimum block size to use for leak detection.

`output_dir=output_data_dir`
: Species the name of the output data directory.

`scramble_blocks=on|off (or) set heap-check scramble <on/off>`
: Enables block scrambling.

Batch mode leak detection stops the application at the end, when libraries are being unloaded, and invokes HP WDB to print the leak or heap data.

Note:

It is incorrect usage to use spaces before or after the '=' symbol in the batch mode configuration options in the configuration file, `rtcconfig`. Additionally, it is incorrect usage to use spaces before the batch mode configuration options.

For example:

Correct Usage:

```
$ cat rtcconfig
check_leaks=on
check_heap=on
files=batchrtc4
$
```

Incorrect Usage:

```
$ cat rtcconfig
 check_leaks=on
check_heap = on
files=batchrtc4
$
```

Steps for Batch Mode Memory Debugging

To use batch memory debugging, complete the following steps:

1. Compile the source files
2. Create an `rtcconfig` file in the current directory.
3. Define an environment variable `BATCH_RTC`. If you are using the Korn or Posix shell, enter the following command at the HP-UX prompt:

   `export BATCH_RTC=on`
4. Complete one of the following steps to preload the `librtc` runtime library:
   - Set the target application to preload `librtc` by using the `+mem_check` option for the `chatr` command. In addition to automatically loading the `librtc` library, the `+mem_check` option for the `chatr` command also maps the shared libraries as private.

     To enable or disable the target application to preload the `librtc` runtime library, enter the following command at the HP-UX prompt:

     ```
     $ chatr +mem_check <enable|disable> <executable>
     ```

     Note: The `chatr +mem_check` option preloads the `librtc` runtime library from the following default paths:

     - For 32 bit IPF applications,

       `/opt/langtools/lib/hpux32/librtc.so`
     - For 64 bit IPF applications,

       `/opt/langtools/lib/hpux64/librtc.so`
     - For 32 bit PA applications,

       `opt/langtools/lib/librtc.sl`
     - For 64-bit PA applications,

       `/opt/langtools/lib/pa20_64/librtc.sl`

     To preload the `librtc` runtime library from a path that is different from the default paths, you must use the `LD_PRELOAD` environment variable.

     `(Or)`
   - Instead of automatically preloading `librtc` and mapping the shared libraries, you can explicitly preload the required `librtc` library after mapping the shared libraries private.

     In the case of HP 9000 systems, you must explicitly map the share libraries as private by using the `+dbg enable` option for the `chatr`command, as follows:

     `$ chatr +dbg enable ./<executable>`

(This step is not required on Integrity systems.)

To explicitly preload the `librtc` runtime library and start the target application, enter one of the following commands:

- For 32 bit IPF applications,

  `LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable>`
- For 64 bit IPF applications,

  `LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable>`
- For 32 bit PA applications,

  `LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable>`
- For 64-bit PA applications,

  `LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable>`

If `LD_PRELOAD` and `chatr +mem_check` are used to preload the `librtc` runtime library , the `librtc` runtime library is loaded from the path specified by `LD_PRELOAD`.

Note:

Batch Mode RTC displays one of the following errors and causes the program to temporarily hang if the version of WDB and `librtc.[sl|so]` do not match, or if WDB is not available on the system:

```
"/opt/langtools/bin/gdb: unrecognized option ‘-brtc’
Use ‘/opt/langtools/bin/gdb --help’ for a complete list of options."
```

(OR)

```
"execl failed. Cannot print RTC info: No such file or directory"
```

This error does not occur under normal usage where WDB or `librtc.[sl|so]` is used from the default location at `/opt/langtools/...` However, this error occurs if `GDB_SERVER` and/or `LIBRTC_SERVER` are set to a mismatched version of WDB or `librtc.[sl|so]` respectively.

5. At the end of the run, output data file is created in `output_data_dir`, if defined in `rtcconfig`,or the current directory. HP WDB creates output data file for each run. It creates a separate file for leak detection and heap information. The naming convention for output files is as follows:

   `<file_name>.<pid>.<suffix>`

   Where, <pid> is the process id and the value for <suffix> can be either `leaks`, `heap`, or `mem`.

   Note:

   During operations such as `system(3s)`and `popen()`, which invoke a new shell, `librtc.sl|so` must not be loaded to the invoked shell. You must use `LD_PRELOAD_ONCE`, instead of `LD_PRELOAD`, to exclusively load the `librtc.sl|so` file to the calling process only. Following is the syntax for using `LD_PRELOAD_ONCE`:

   `LD_PRELOAD_ONCE= /opt/langtools/lib/librtc.sl`

14.10.5.2 Environment variable setting for Batch mode debugging

Batch mode memory leak detection uses the following environment variables:

- `GDBRTC_CONFIG` species the location of `rtc` configuration file. If this option is not specified, the configuration file is assumed to be in the current location, and has the filename `rtcconfig`. If user prefers to set this option, it must include the filename.
  - Incorrect usage:
    ```
    export GDBRTC_CONFIG=./
    export GDBRTC_CONFIG=/tmp
    ```
  - Correct usage:
    ```
    export GDBRTC_CONFIG=/tmp/yet_another_config
    export GDBRTC_CONFIG=/tmp/rtcconfig
    ```
- `BATCH_RTC` enables or disables batch memory leak detection.
- `GDB_SERVER` is used to override the default path from where the `gdb` executable is used to provide the information on memory leak. By default, `/opt/langtools/bin/gdb` is used to print the output. This can be overriden by setting `GDB_SERVER` appropriately.
- `RTC_MALLOC_CONFIG` is used to override the default `config` and `rtcconfig` file settings. This variable can be set as follows:
  ```
  export RTC_MALLOC_CONFIG=config_string1[;config_strings].
  ```
  - The `config_strings` are separated by ;.
  - The following config_strings options exist for `RTC_MALLOC_CONFIG`:

    `abort_on_bounds=[01]`
    : Aborts execution when heap objects bounds check fail, value is 1, and the environment variable `RTC_NO_ABORT` is not set.

    `abort_on_bad_free=[01]`
    : Aborts execution when free or `realloc` is trying to free a heap object which is not valid, value is 1, and environment variable `RTC_NO_ABORT` is not set.

    `abort_on_nomem=[01]`
    : Aborts execution when out of memory if value is 1, and environment variable `RTC_NO_ABORT` is not set.

    `leak_logfile=stderr[+]filename`
    : The log file for batch mode must be specified.

      `stderr`: error message goes to `stderr`

      `[+]filename`: error message goes to filename, `+` means output is appended to the file.

      ```
      mem_logfile=stderr[+]filename
      heap_logfile=stderr[+]filename
      ```

  - Specify `config_strings` for `+check=malloc` on Itanium or WDB memory check batch mode on Integrity systems.

14.10.5.3 Example for Batch Mode RTC

This section illustrates examples of batch mode RTC on HP 9000 and Integrity Systems.

- Example for Batch Mode RTC of a PA-RISC 32 bit Executable

  Step 1: Compile the source files.

  Step 2: The `rtcconfig` file must entries, such as the following:

  ```
  check_heap=on
  check_leaks=on
  check_free=on
  files=executable_name
  output_dir=/tmp/results
  ```

  Step 3: Set the following environment variables as follows:

  ```
  export BATCH_RTC=on
  ```

  Step 4: Complete one of the following steps:

  - Map the shared libraries privately using `chatr`, as follows:

    ```
    chatr +dbg enable <executable>
    ```

    Explicitly preload the librtc runtime library and start the target application, as follows:

    ```
    LD_PRELOAD=/opt/langtools/lib//librtc.sl <executable>
    <arguments>
    ```

    `(Or)`

  - Set the target application to preload `librtc` by using the `+mem_check` option for the `chatr` command. In addition to automatically loading the `librtc` library, the `+mem_check` option for the `chatr` command also maps the shared libraries as private.

    To set the target application to preload the `librtc` runtime library, enter the following command at the HP-UX prompt:

    ```
    $ chatr +mem_check enable <executable>
    ```

- Example for Batch Mode RTC on Integrity Systems: Example-1

  Step 1: Compile the source files.

  Step 2: The `rtcconfig` file must contain entries such as the following:

  ```
  check_heap=on
  check_leaks=on
  check_free=on
  files=executable_name
  output_dir=/tmp/results
  ```

  Step 3: Set the following environment variables as follows:

  ```
  export BATCH_RTC=on
  ```

  Step 4: Complete one of the following steps:

  - Preload the `librtc` runtime library, as follows: `LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable> <arguments if any>`

    (Or)

  - Preload the `librtc` runtime library, as follows: `$ chatr +mem_check enable <executable>`

- Example from Integrity Systems: Example-2

  Step 1: Compile the source files.

  Step 2: The `rtcconfig` file should contain entries such as the following:

  ```
  check_heap=on
  check_leaks=on
  check_free=on
  files=exec1:exec2:exec3
  output_dir=/tmp/results
  ```

  Step 3: Set the following environment variables as follows:

  ```
  export BATCH_RTC=on
  ```

  Step 4: Complete one of the following steps:

  - Use the `+mem_check` option for the `chatr` command on each of the required executable files that must be instrumented, as follows:

    ```
            $ chatr +mem_check enable exec1 exec2 exec3
    ```

    (Or)

  - Preload the `librtc` library as follows:

    ```
    export LD_PRELOAD /opt/langtools/lib/hpux32/librtc.so
    ```

  Step 5: Run the program as follows:

  ```
  ./exec1
  ```

  Suppose `exec1` eventually spawns `exec2`, `exec3`, `exec4`, `exec5`, only `exec1`, `exec2`, `exec3` will be instrumented based on the settings in the `rtcconfig` file.

14.10.6 Debugging Memory Interactively After Attaching to a Running Process

HP WDB accepts `-pid` or `-p` followed by a process ID to attach a running process to the debugger.

To use Batch RTC after attaching GDB to a running process, complete the following steps:

1. Complete one of the following steps to preload the `librtc` runtime library:
   - Set the target application to preload `librtc` by using the `+mem_check` option for the `chatr` command.In addition to automatically loading the `librtc` library, the `+mem_check` option for the `chatr` command also maps the shared libraries as private.

To enable or disable the target application to preload the `librtc` runtime library, enter the following command at the HP-UX prompt:

```
$ chatr +mem_check <enable|disable> <executable>
```

Note: To attach and find leaks for PA-32 applications from the startup, the environment variable `RTC_INIT` should be set to `on` in addition to preloading the `librtc.[sl|so]` library before starting the application, as follows:

```
$ RTC_INIT=on <executable>
                    (Or)
```

- Instead of automatically preloading `librtc` and mapping the shared libraries, you can explicitly preload the required `librtc` library.

  In the case of HP 9000 systems, you must explicitly map the share libraries as private by using the `+dbg enable` option for the `chatr` command, as follows:

  `$ chatr +dbg enable ./<executable>` (This step is not required on Integrity systems.)

  To explicitly preload the `librtc` runtime library and start the target application, enter one of the following commands:

  - For 32 bit IPF applications,

    `LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable>`

  - For 64 bit IPF applications,

    `LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable>`

  - For 32 bit PA applications,

    `LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable>`

  - For 64-bit PA applications,

    `LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable>`

  Note: To attach and find leaks for PA-32 applications from the startup, the environment variable `RTC_INIT` should be set to `on` in addition to preloading the `librtc.[sl|so]` library before starting the application, as follows:

  ```
  $ LD_PRELOAD=/opt/langtools/lib/librtc.sl RTC_INIT=on <application>
  ```

  If `RTC_INIT` is turned `on`, `librtc` starts recording heap information for PA32 process, by default. Hence, you must set this environment variable only when it is required, and you must not export this environment variable for shell.

2. Run the program.
3. Start a debugging session as follows:

   ```
   gdb -leaks <executable-name> <process-id>
   ```

4. Use `info heap` and `info leaks` commands to obtain a memory analysis report of the application.

Note: From HP WDB 5.7 onwards, the archive version of the run time check library, `librtc.a`, is not available. You must use the shared version of the library, `librtc.[sl|so]`, instead.

14.10.7 Configuring memory debugging settings

The following configuration settings are supported to control the level of details of information required to be displayed when debugging memory leaks.

14.10.7.1 Specifying the stack depth

Memory debugging reduces the performance of an application by 20-40% because of stack unwinding. To provide a clear profile of every allocation in the program, the debugger collects the stack trace information for every allocation in the debugged application. Reducing the stack depth (the number of stack frames that the debugger collects for each allocation) reduces the performance degradation.

The `set heap-check frame-count` command enables you to control the depth of the stack frames that are collected by WDB for each allocation. By default, four stack frames are displayed from the allocating call stack.

To set the depth of the stack frames that is collected by WDB, enter the following command at the gdb prompt:

`$ set heap-check frame-count [num]`

The stack depth, `[num]`, is the number of stack frames that WDB collects for each allocation.

You can specify a higher value for `[num]` to view more stack frames for each reported allocation. However, the performance of the application is reduced because of the increased stack depth.

14.10.7.2 Specifying minimum leak size

WDB enables you to specify the minimum leak size for stack trace collection to improve the program's performance.

Stack trace collection slows down the program because it occurs on every allocation call. Therefore, if the program is allocation-intensive, WDB can spend a substantial amount of time collecting stack traces.

You can improve performance by using the command:

```
set heap-check min-leak-size num
```

For example, if you use,

```
set heap-check min-leak-size 100
```

WDB does not collect stack traces for allocations smaller than 100 bytes. HP WDB still reports leaks smaller than this size, but does not include a stack trace.

14.10.7.3 Specifying minimum block size

The `min-heap-size` option reports the heap allocations that exceed the specified number `<num>` of bytes based on the cumulative number of bytes that are allocated at each call-site, which is inclusive of multiple calls to `malloc` at a particular call site.

Example: set heap-check min-heap-size 100

When the option `min-heap-size` is set to 100, GDB reports all the cumulative block allocations that 100 bytes at each call-site.

14.10.8 Scenarios in memory debugging

14.10.8.1 Stop when freeing unallocated or deallocated blocks

WDB enables you to locate improper calls to `free()` and `realloc()`, with the command

- In interactive debugging mode: `set heap-check free [on | off]`.
- In batch mode debugging: `check_free [on | off]`.

With this setting on, whenever the program calls `free()` or `realloc()` WDB inspects the parameters to verify that they are pointing to valid currently allocated heap blocks.

If WDB detects an erroneous call to `free()`, it stops the program and reports this condition. You can then look at the stack trace to understand where and how the problem occurred.

14.10.8.2 Stop when freeing a block if bad writes occurred outside block boundary

WDB enables you to locate problems caused by heap block overflow or underflow with the command

- In Interactive debugging mode: `set heap-check bounds [on | off]`
- In batch mode debugging: `check_bounds [on | off]`.

When bounds checking is turned `on`, WDB allocates extra space at the beginning and end of a block during allocation and fills it up with a specific pattern. When blocks are freed, WDB verifies whether these patterns are intact. If they are corrupted, an underflow or overflow must have occurred and WDB reports the problem. If you want to find corruption at any time, use the `info corruption` command.

The `info corruption` command can detect memory corruption in an application. That is, it reports all the memory blocks that have over-writes and under-writes.

Syntax:

```
info corruption [<file name>]
```

The run time memory checking must be enabled before using the `info corruption` command to detect memory corruption. The corruption information is written to a file specified in the .file name argument if provided. Otherwise, it is printed to the `stdout`.

> *Note:* Turning on bounds checking increases the program's memory requirements because the extra guard bytes must be allocated at the beginning and end of each block.

14.10.8.3 Stop when a specified block address is allocated or deallocated

To stop a program whenever a block at a specified address is allocated or deallocated, use the command:

```
set heap-check watch address
```

You can use this to debug situations such as multiple `free()` calls to the same block.

Limitation : This is not supported in batch mode debugging.

14.10.8.4 Scramble previous memory contents at malloc/free calls

WDB enables you to potentially expose latent memory access defects with the command:

- In Interactive debugging mode: `set heap-check scramble [on | off]`
- In batch mode debugging: `scramble_blocks [on | off]`.

When this setting is turned `on`, any time a memory block is allocated or deallocated, WDB scrambles the space and overwrites it with a specific pattern.

This change to the memory contents increases the chance that erroneous behaviors will cause the program to fail. Examples of such behavior include attempting to access space that is freed or depending on initial values of `malloc()` blocks.

You can now look at the stack trace to understand where and how the problem occurred.

> *Note:* Turning on scrambling slows down the program slightly, because at every `malloc()` and `free()` call, the space involved must be overwritten.

14.10.8.5 Detect dangling pointers and dangling blocks

A pointer is a *Dangling pointer* if the block of memory it points to, has been freed by the application. The block is called *Dangling Block*.

The same freed block could be subsequently allocated to the application in response to another memory allocation request. In this scenario, if the application incorrectly tries to write into the freed memory block using the dangling pointer, it could result in incorrect or an undefined program behavior, as the new owner or function owning the same allocated block would find different values in the heap block.

Note:

Software literature names this concept as *premature free* or *Reading/writing freed memory using a pointer*.

WDB tracks the dangling pointers and dangling blocks using a modified version of Garbage collection. The enabler for doing this is by retaining all the freed blocks internally within RTC without actually freeing it as long as possible. It displays all the potential pointers to the freed dangling blocks, in the application data space.

The pointers are *potential* because the pointers need not be actual pointers and could be a *datum* value and hence there are chances of *false positives* in the dangling report.

Note:

WDB tries to help as much as possible to detect if these pointers are of type datum or real pointers. In a `-g` compiled binary, WDB performs a look-up on a symbol table to find the symbol name and type to find the symbol name of the potential pointer and if its of pointer type, then the corresponding dangling block is really dangling(not a false positive).

WDB turns on these checks, only when you specify `set heap-check retain-freed-blocks on`.

14.10.8.6 Detect in-block corruption of freed blocks

HP WDB detects all the attempts of a program to write to the freed dangling blocks using dangling pointers.We detect such in-block corruptions and are reported as part of the existing `info corruption` command output.

14.10.8.7 Specify the amount of guard bytes for every block of allocated memory

HP WDB enables you to programmatically control the size of guard bytes for every block of the allocated memory. You can use these guard bytes to spot very rare and non-trivial boundary (buffer over-run and buffer under-run) corruptions. This again is available optionally when the user specifies `set heap-check retain-freed-blocks <on>`.

14.10.9 Comparison of Memory Debugging Commands in Interactive Mode and Batch Mode

HP WDB 5.6 and later versions provide consistency in format for the batch mode options and the interactive mode commands.

The following table lists the memory debugging commands available in batch and interactive mode:

Memory Debugging Commands in Interactive and Batch Mode

Command Description	**Interactive mode**	**Batch mode**
Toggles heap profiling and detection of leaks, bounds, and double free	`set heap-check [on | off]`	`check_heap= [on | off]` (or) `set heap-check [on | off]`
Toggle the leak detection capability	`set heap-check leaks [on |off]`	`check_leaks =[on | off]` (or) `set heap-check leaks [on |off]`
Toggle validation of calls to `strcpy`, `strncpy`, `memcpy`, `memccpy`, `memset`, `memmove`, `bzero` and, `bcopy`	`set heap-check string [on | off]`	`check_string= [on |off]` (or) `set heap-check string [on | off]`
Toggle validation of calls to `free()`	`set heap-check free [on |off]`	`check_free = [on |off]` (or) `set heap-check free [on | off]`
Specify whether freed blocks should be scrambled	`set heap-check scramble [on |off]`	`scramble_blocks=[on| off]`(or) `set heap-check scramble [on |off]`
Toggle bounds check on heap blocks	`set heap-check bounds [on| off]`	`check_bounds = [on | off]` (or) `set heap-check bounds [on| off]`
Specify the minimum size of a block for stack trace collection. GDB will report blocks of size greater than or equal to `<num>` at each call-site.	`set heap-check min-heap-size <num>`	`min_heap_size = <num>` (or) `set heap-check min-heap-size <num>`

Specify the minimum size of a block for stack trace collection. GDB report leaks of blocks, smaller than this value. However, no stack trace is provided.	`set heap-check min-leak-size <num>`	`min_leak_size = <num>` (or) `set heap-check min-leak-size <num>`
Specify the depth of call stack to be captured	`set heap-check frame-count <num>`	`frame_count = <num>` (or) `set heap-check frame-count <num>`
Instruct GDB to stop, whenever a block at the given address gets allocated or deallocated.	`set heap-check watch address`	Not supported in batch mode
Instructs `malloc` to return null after <num> invocations of `malloc`.	`set heap-check null-check <num>`	Not supported in batch mode
Instructs `malloc` to return null after <num> bytes have been allocated by `malloc`.	`set heap-check null-check-size <size>`	Not supported in batch mode
Specifies the seed value to be used for generating random `null-check-count`	`set heap-check seed-value <num>`	Not supported in batch mode
Specifies the random range to be used by random range	`set heap-check random-range <num>`	Not supported in batch mode
Specifies the time interval to be used for incremental memory profile	`set heap-check interval <num>`	Not supported in batch mode
Perform incremental profile for <num> interval periods where each period duration is defined by set heap-check interval command. The default value is 100.	`set heap-check repeat <num>`	Not supported in batch mode

NULL pointer return by memory allocators; used with set heap-check on, with/without null-check enabled	`catch nomem`	Not supported in batch mode

Note:

> The `files=<executable-name>` and `output_dir=<path>` options are exclusive to batch mode debugging.

14.10.10 Heap Profiling

The heap profile is useful for identifying how memory is being used by the program. You can use WDB to profile an application's current heap usage.

14.10.10.1 Commands for heap profiling

`info heap`
: Displays a heap report, listing information such as the start of the heap, end of the heap, heap allocations, size of blocks, and number of instances. The report shows heap usage at the point you use the `info heap` command.

  The report does not show allocations that have already been freed. For example, if you make several allocations, free them all, and then you use `info heap`, the result does not show any allocations.

`info heap` *`filename`*
: Writes heap report output to the specified file.

`info heap` *`idnumber`*
: Produces detailed information on the specified heap allocation including the allocation call stack.

`set heap-check frame-count` *`num`*
: Controls the depth of the call stack collected. Larger values increase run time. The default value is four (4) stack frames.

`show heap-check`
: Displays all current settings for memory checking.

14.10.10.2 `info heap arena`

The `info heap arena` command enables the user to view high level memory usage details of each arena. The `info heap arena` is not supported in batch mode. This command is available only for applications running on 11i v3 or later.

14.10.10.3 `info heap arena [0 |1|2|..] blocks stacks`

Displays the memory profile report for block level and overall memory usage with stack trace where applicable.This command is available only for applications running on 11i v3 or later.

14.10.10.4 `info module ADDRESS`

The `info module` command identifies load modules, and determines whether it lies in the text or data region for a given address. This command is available only for applications running on 11i v3 or later.

Syntax:

`info module ADDRESS`

14.10.10.5 `info heap process`

The `info heap process` command enables the user to view a high level memory usage report of a process. This command is available only for applications running on 11i v3 or later.

14.10.10.6 Example for heap profiling

This example shows how to use this feature on HP-UX 11.x:

1. If the linker version is earlier than B.11.19, link with `/opt/langtools/lib/pa20_64/librtc.sl` for PA-64 programs. For a 32-bit program, you must link with '`/opt/langtools/lib/librtc.sl`'.

   If the dynamic linker version is B.11.19 or later, skip this step because WDB automatically loads the '`librtc.sl`' library.
2. Turn on profiling with the `set heap-check on` command:
   ```
   (gdb) set heap-check on
   ```
3. Set a breakpoint:
   ```
   (gdb) b myfunction
   ```
4. When the program is stopped at a breakpoint, use the `info heap` command:
   ```
   (gdb) info heap

   Analyzing heap ...done

   Actual Heap Usage:
   Heap Start = 0x40408000
   Heap End = 0x4041a900
   Heap Size = 76288 bytes

   Outstanding Allocations:
   41558 bytes allocated in 28 blocks
   ```

```
No.   Total bytes     Blocks     Address     Function
0        34567          1       0x40411000   foo()
1        4096           1       0x7bd63000   bar()
2        1234           1       0x40419710   baz()
3        245            8       0x404108b0   boo()

[...]
```

The display shows the currently allocated heap blocks. Any blocks that have been allocated and already freed are not listed.

To look at a specific allocation, specify the allocation number with the `info heap` command:

```
(gdb) info heap 1
4096 bytes at 0x7bd63000 (9.86% of all bytes allocated)
  in bar () at test.c:108
  in main () at test.c:17
  in _start ()
  in $START$ ()
```

When multiple blocks are allocated from the same call stack, WDB displays additional information:

```
(gdb) info heap 3
245 bytes in 8 blocks (0.59% of all bytes allocated)
These range in size from 26 to 36 bytes and are allocated
  in boo ()
  in link_the_list () at test.c:55
  in main () at test.c:13
  in _start ()
```

14.10.11 Memory Checking Analysis for User Defined Memory Management Routines

Many user applications have their own memory management routines. These custom allocator routines are either user defined or sometimes wrappers to the default memory management routines. Memory checking features has been extended for user defined memory management routines. Memory leak, profile of heap memory or memory errors can be obtained for user defined memory management routines.

Limitations:

- This feature routes to default memory routines instead of calling user defined memory management routines to obtain memory analysis.
- This feature is not supported in batch mode and attach mode debugging.

14.10.12 Commands to track the change in data segment value

The high water mark records the number of times the `brk()` value changes. The following commands are supported:

`info heap high-mem`
: Displays the number of times `brk()` value changes for a given run.

`set heap-check high-mem-count X_number`
: Stops when `brk()` value has moved `X_number` of times.

Limitations:

- This feature assumes that an application has a deterministic memory allocation pattern from one run to another.
- The `high_mem` feature is not supported in batch mode debugging.

14.11 Thread Debugging Support

HP WDB provides thread-debugging support for kernel, user, and MxN threads. You can exclusively disable or enable specific thread execution. Advanced thread debugging support in HP WDB enables you to view information on pthread primitives and detect certain thread-related conditions.

> Note: WDB supports `pthread` parallelism, but it does not support compiler-generated parallelism such as parallelism with directives.

14.11.1 Support for Enabling and Disabling Specific Threads

When you suspect that a specific thread is causing a problem while debugging a multi-threaded application, you can suspend the execution of other threads in the application and debug the doubtful thread exclusively. HP WDB 3.2 and later versions provide the following commands to disable and enable specific thread execution:

- `thread disable` - This command prevents the specified threads from running until they are enabled again using the `thread enable` command.
- `thread enable` - This command enables the specified thread to run when you issue the `continue` or `step` command. By default, all threads are in the enabled state. You can use the `thread enable` command to reactivate a disabled thread.

Consider the following example:

```
(gdb) info thread
system thread 4189  0x7f666da8
in __pthread_create_system+0x3d8 () from /usr/lib/libpthread.1
2 system thread 4188  worker (wptr=0x40004640 "̈) at quicksort.c:135
1 system thread 4184  0x7f66f728 in _lwp_create+0x10 () from /usr/lib/libpthread.1
```

To disable a thread, execute the following command:

```
(gdb) thread disable 1
warning: ATTENTION!! Disabling threads may result in
deadlocks in the program.Disabling thread 1
```

To enable a thread, execute the following command:

```
(gdb) thread enable 1
Enabling thread 1
```

14.11.2 Backtrace Support for Thread Debugging

The following commands are available as backtrace support for thread debugging:

- `bt`

  The `bt` command provides the stack trace of the current thread that is being executed or the thread that accepts the signal in case of a core file.

- `thread apply all bt`

  You can use the `thread apply all bt` command to display the backtrace of all threads. The `bt` command only provides the stack trace of the current thread under execution.

- `backtrace_other_thread`

  The `backtrace_other_thread` command prints the backtrace of all stack frames for a thread with stack pointer SP, program counter PC and address of `gr32` in the backing store BSP. This command enables you to view the stack trace when the stack is corrupted. When using this command, you must ensure that the SP, PC, and BSP values are valid.

  The syntax for the `backtrace_other_thread` command is as follows:

  ```
  backtrace_other_thread SP PC BSP
  ```

14.11.3 Advanced Thread Debugging Support

Advanced thread debugging support is available for multi-threaded applications running on HP-UX 11iv2, or HP-UX 11iv3.

HP WDB 5.5 and later versions provide advanced thread debugging features to display extended information on the state of `pthread` primitives such as mutexes, read-write locks and conditional variables.

HP WDB 5.6 and later versions provide advanced thread-debugging options to detect the following thread-related conditions:

- The thread attempts to acquire a non-recursive mutex that it currently holds.
- The thread attempts to unlock a mutex or a read-write lock that it has not acquired.
- The thread waits (blocked) on a mutex or read-write lock that is held by a thread with a different scheduling policy.
- Different threads non-concurrently wait on the same condition variable, but with different associated mutexes.
- The thread terminates execution without unlocking the associated mutexes or read-write locks.
- The thread waits on a condition variable for which the associated mutex is not locked.

- The thread terminates execution, and the resources associated with the terminated thread continues to exist in the application because the thread has not been joined or detached.
- The thread uses more than the specified percentage of the stack allocated to the thread.
- The number of threads waiting on any pthread object exceeds the specified threshold number.

14.11.3.1 Pre-requisites for Advanced Thread Debugging

The following pre-requisites apply for advanced thread debugging:

- The advanced thread debugging features are supported only on HP-UX 11i v2 and later on both PA-RISC and Integrity systems.
- The tracing `pthread` library is required for advanced thread-debugging. The `pthread` tracer library is available by default on systems running on HP-UX 11i v2 or later. You must install HP WDB 5.5 or later versions of the debugger to support enhanced thread debugging. The installation scripts for HP WDB 5.5 and later versions of the debugger automatically add links at `/opt/langtools/lib/` to replace the standard `libpthread` library with `libpthread tracer` library at run time.
- The thread debugging feature in HP WDB is dependent on the availability of the dynamic Linker Version B.11.19.
- HP WDB uses `librtc.sl` to enable thread debugging support. If the debugger is installed in a directory other than the default `/opt/langtools/bin` directory, you must use the environment variable, `LIBRTC_SERVER`, to export the path of the appropriate version of `librtc.sl`.
- HP WDB does not support debugging of programs that link with the archive version of the standard C library `libc.a` or the core library `libcl.a`. The programs must be linked with `libc.sl`.
- The advanced thread debugging commands work only if `set thread-check` is set to on.
- For PA-RISC 32-bit applications, the dynamic library path look-up must be enabled for advanced thread debugging. To enable dynamic library path look-up for advanced thread debugging, enter the following command at HP-UX prompt:

  ```
  # chatr +s enable <PA32-bitApp>
  ```

  This command automatically enables dynamic library path look-up. No additional environmental variables are required to be set.

14.11.3.2 Enabling and Disabling Advanced Thread Debugging Features

HP WDB 5.6 and later versions of the debugger provide advanced thread debugging features for debugging multi-threaded applications running on 11i v2, or 11i v3. These features are available as options to the `set thread-check` command. The syntax for the `set thread-check` command is as follows:

```
set thread check { [on|off]| [option] [on|off] |[option] [num]}
```

The `set thread-check [on|off]` command enables or disables advanced thread debugging. This feature is `off` by default. The `set thread-check [on|off]` command must be enabled prior to running the application under the debugger, to force the underlying runtime system to collect information on `pthread` primitives.

The advanced thread debugging features can be enabled only if the `set thread-check [on]` command is enabled. The following advanced thread debugging options are available for the `set thread-check` command:

- `recursive-relock [on|off]`

  This `set thread-check recursive-relock [on|off]` command checks if a thread has attempted to acquire a non-recursive mutex that it currently holds. Re-locking a non-recursive mutex results in a deadlock. At run-time, the debugger keeps track of each mutex in the application and the thread that currently holds each mutex. When a thread attempts to acquire a lock on a non-recursive mutex, the debugger checks if the thread currently holds the lock object for the mutex. The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

- `unlock-not-own [on|off]`

  The `set thread-check unlock-not-own [on|off]` command checks if a thread has attempted to unlock a mutex or a read-write lock that it has not acquired. The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

- `mixed-sched-policy [on|off]`

  The `set thread-check mixed-sched-policy [on|off]` command checks if a thread is waiting (blocked) on a mutex or a read-write lock that is held by a thread with a different scheduling policy. This is not an application error and does not always result in deadlock. However, it degrades the application performance. The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

- `cv-multiple-mxs [on|off]`

  The `set thread-check cv-multiple-mxs [on|off]` command checks if an application uses the same condition variable in multiple calls, by different threads to `pthread_cond_wait()` or `pthread_cond_timedwait()`, but specifies different mutexes. The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

  All threads that concurrently wait on any single condition variable must specify the same associated mutex. For example,the `pthread` implementation does not allow `thread 1` to wait on `condition variable A` by specifying `mutex A`, while `thread 2` waits on the same `condition variable A` by specifying `mutex B`. This returns an `EINVAL` error to the application. In contrast to this obvious `EINVAL` error, the `cv-multiple-mxs` option detects the less-obvious and non-concurrent use of multiple mutexes with the same condition variable by different threads. This option detects potential `EINVAL` errors that exist as a result of different timings or execution paths.

This option is used to detect unintentional use of multiple mutexes with the same condition variable by different threads. In the case of applications that use a dynamic pool of mutexes, the `cv-multiple-mxs` option is not required because this usage is normal and expected application behavior.

- `cv-wait-no-mx [on|off]`

  The `set thread-check cv-wait-no-mx [on|off]` checks if the associated mutex of a condition variable is locked when the thread calls the `pthread_cond_wait()` routine. The debugger transfers the execution control to the user and prints a warning message when this condition is detected. This check is not a `POSIX.1` standard requirement for the `pthread_cond_wait()` routine. It is an additional check provided by WDB.

- `thread-exit-own-mutex [on|off]`

  The `set thread-check thread-exit-own-mutex [on|off]` command checks if any thread has terminated execution without unlocking the mutexes or read-write locks that are associated with the thread. The debugger transfers the execution control to the user and prints a warning message when this condition is detected. This situation can result in deadlocks if other threads are waiting to acquire the locked mutexes or read-write locks.

- `thread-exit-no-join-detach [on|off]`

  The `set thread-check thread-exit-no-join-detach [on|off]` command checks if a thread has terminated execution (either successfully or because of an exception or a cancel) without joining or detaching the thread. The debugger transfers the execution control to the user and prints a warning message when this condition is detected. The resources associated with a terminated thread continue to exist in the application if the thread is not joined or detached. The thread must be explicitly joined or detached, or it must be created with the detach attribute. When an application repeatedly creates threads without a join or detach operation, the application leaks resources. This may result in application failure.

- `stack-util [num]`

  The `set thread-check stack-util [num]` command checks if any thread has used more than the specified percentage `[num]` of the stack allocation. The debugger transfers the execution control to the user and prints a warning message when this condition is detected. You can use this option to verify if there is a margin of safety in stack utilization by the threads. The application must ensure that the thread stack size is sufficient for all the operations of the thread. Each thread is assigned a stack allocation when it is created. If the stack allocation is not specified for a thread, the default value is used. The stack allocation cannot be modified after a thread is created. If a thread attempts to use more stack space than the allocated stack space, it results in a stack overflow. Stack overflow can result in memory access violations, bus errors, or segmentation faults.

- `num-waiters [num]`

  The `set thread-check num-waiters [num]` command checks if the number of threads waiting on any `pthread` object exceeds the specified threshold number `[num]`. The debugger transfers the execution control to the user and prints a warning message when this condition is detected.

14.11.3.3 Commands to view information on pthread primitives

WDB 5.5 and later versions of the debugger display extended information on `pthread` primitives for multi-threaded applications running on HP-UX 11i v2, or 11i v3. This feature is available only if the `set thread-check [on|off]` command is enabled. The following commands enable you to view extended information on threads, mutexes, read-write locks and conditional variables in multi-threaded applications:

- `info thread [thread-id]`

  The `info thread [thread-id]` command displays a list of known threads. If a `thread-id` is provided, the command displays extended information on the specified thread.
- `info mutex [mutex-id]`

  The `info mutex [mutex-id]` command displays a list of known mutexes. If a `mutex-id` is provided, the command displays extended information on the specified mutex.
- `info condvar [condvar-id]`

  The `info condvar [condvar-id]` command displays a list of known condition variables. If `condvar-id` is provided, the command displays extended information on the specified condition variable.
- `info rwlock [rwlock-id]`

  The `info rwlock [rwlock-id]` command displays a list of known read-write locks. If `rwlock-id` is provided, the command displays extended information on the specified read-write lock.

14.11.4 Debugging Threads Interactively After Attaching to a Process

HP WDB provides support to attach a running process to the debugger. To use thread debugging commands after attaching GDB to a running process, complete the following steps:

1. Set `LD_LIBRARY_PATH` to include the appropriate directory, by entering one of the following commands:
   - For 32 bit IPF applications,

     `LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux32`
   - For 64 bit IPF applications,

     `LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux64`
   - For 32 bit PA applications,

     `LD_LIBRARY_PATH=/opt/langtools/wdb/lib`

- For 64-bit PA applications,

  `LD_LIBRARY_PATH=/opt/langtools/wdb/lib/pa20_64`

2. Explicitly preload the `librtc` runtime library and start the target application by entering one of the following commands:
   - For 32 bit IPF applications,

     `LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable>`
   - For 64 bit IPF applications,

     `LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable>`
   - For 32 bit PA applications,

     `LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable>`
   - For 64-bit PA applications,

     `LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable>`
3. Complete one of the following steps:
   - Attach the debugger to the required process and enable thread debugging, as follows:

     ```
     gdb -thread -p <pid>
            or
     gdb -thread <executable> <pid>
     ```
   - Alternately, you can attach the process to the debugger and consequently invoke thread debugging, as follows:

     ```
     $ gdb <executable> <pid>
     ...
     (gdb)set thread-check on
     ```

14.11.5 Thread Debugging in Batch Mode

HP WDB supports batch mode of debugging threads for HP-UX 11iv2 and later, on Integrity systems and on HP-UX 11i v3 in PA-RISC systems for 64 bit applications. The debugger provides a log file with the list of thread-related errors that occur in the application.

In batch mode, the debugger detects the all the thread-conditions that are detected during an interactive debugging session.

The debugger reports extended information such as variable address, name, id and other specifications related to the involved pthread objects. In addition, it displays the stack trace of the executing thread at the point of error.

Note:

Use the `set frame-count` setting in the `rtconfig` file to control the depth of the stack trace file. This command controls the depth of the call stack collected. Larger values increase the run time.

14.11.5.1 Pre-requisites for Batch mode of Thread Debugging

The various prerequisites for Batch mode of Thread Debugging are as follows:

- The thread-debugging feature in HP WDB is dependent on the availability of the dynamic linker version B.11.19.
- Advanced thread-debugging feature requires the pthread tracer library which is available by default on systems running HP-UX 11i v2 or later.

Steps to debug threads in batch mode

1. Compile the source files.
    - Set the `LD_LIBRARY_PATH` environment variable, based on the platform as follows:
    - For IPF 32 bit applications, set
      `LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux32`
    - For IPF 64 bit applications, set
      `LD_LIBRARY_PATH=/opt/langtools/wdb/lib/hpux64`
    - For PA 64 bit applications, set
      `LD_LIBRARY_PATH=/opt/langtools/wdb/lib/pa20_64`
2. Map the share libraries as private for HP 9000 systems using the following command:

   `$ chatr +dbg enable ./<executable>`

   Note: This step is not applicable for Integrity systems.
3. Create a configuration file, `rtcconfig` to specify the various thread conditions that you want the debugger to detect.

   Note: The configuration file contains lines of the following form:

   `set thread-check [on|off] | [option] [on|off] | [option] [num]`

   And/Or

   `set frame-count [num]`

   And/Or

   `files=<name of the executable on which the thread checking is to be done>`

   For more information, Section 14.11.3.2 [Enabling and Disabling Advanced Thread Debugging Features], page 157
4. Set the environment variable `BATCH_RTC` to `on` as: `Set BATCH_RTC=on`
5. Preload the librtc runtime library explicitly and start the target application using one of the following commands:
    - For 32 bit IPF applications,
      `LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable>`
    - For 64 bit IPF applications,
      `LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable>`
    - For 64 bit PA applications,
      `LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable>`

If HP WDB detects any thread error condition during the application run, the error log is output to a file in the current working directory. The output file has the following naming convention:

`<executable name>.<pid>.threads`

Where

`<pid>`is the process id.

14.11.5.2 Limitations in Batch mode of thread debugging

The feature does not obtain the thread-error information in batch mode for forked process in a multiprocessing application. However, if the `librtc.sl` library is preloaded, the debugger obtains the thread-error information in the batch mode for `exec`-ed application.

You cannot specify an alternate output directory for the thread-error log. The thread-error log file is output into the current working directory only.

HP WDB cannot execute both batch mode thread check and batch mode heap check together. If the `rtcconfig` file has both entries, then batch heap check overrides the batch thread check.

14.11.6 Known issues with Thread Debugging for Interactive and Batch mode

Issue 1:

During the execution of advanced thread checking for applications that fork, in the interactive mode, the following message appears if the GDB follows the child:

`Pthread analysis file missing!`

This error message appears because the thread-error information for the forked process is not available.

However, if the forked process `exec()`s another binary, the thread-error information is available for the `exec` -ed binary.

Issue 2:

In both interactive and batch modes, if the applications exceed their thread stack utilization, the following error message appears:

`Error accessing memory address`

This occurs when GDB attempts a command line call on an already overflowing thread stack.

14.12 Debugging MPI Programs

You can attach the debugger to Message Passing Interface (MPI) programs for debugging. You must set the one of the following environment variables before you launch the MPI process:

`set MPI_FLAGS= egdb` for invoking GDB

or

`set MPI_FLAGS= ewdb` for invoking WDB

For more information, see the `mpidebug(1)` and `mpienv(1)` manpages.

Attaching the debugger to an MPI process (or to any other process that has not been compiled for debugging) can result in the following warning:

```
warning: reading 'r3' register: No data
```

14.13 Debugging multiple processes (programs with `fork` and `vfork` calls)

14.13.1 Ask mode for `set follow-fork-mode`

The `ask` command prompts the user to select between parent and child as the debugger response to a program call of `fork/vfork`. Based on the user selection, the parent or the child process is debugged.

For example:

```
(gdb) set follow-fork-mode ask (gdb) show follow-fork-mode
```

The debugger response to a program call to `fork` or `vfork` is ask.

```
(gdb) run  Starting program: sample  [New process 4941]  Select follow-fork-
mode: [0] parent [1] child
```

14.13.2 serial mode for `set follow-fork-mode`

The option serial, for the `follow-fork-mode` command, enables debugging of a parent and child process within a debugger session. During a debug session, if the parent process forks a child, the debugger starts debugging the child process. When the child process exits, the debugger switches back to the parent process. The `follow-fork-mode` will work only if there is a `wait()` call by the parent process. This feature is enabled by setting the `follow-fork-mode` flag to `serial`, as specified in the following example:

```
(gdb) set follow-fork-mode serial
```

The `follow-fork-mode` is not supported under following conditions:

- MxN threaded programs
- Parent process is 32-bit and child process is 64-bit and vice versa. For the `follow-fork-mode` to work both parent and child process must be of the same mode.

14.13.3 Support for showing unwind info

The `maint info unwind` command prints the unwind information for the regions unwinded at the given address expression. Usage:

```
maint info unwind exp
```

where `exp` is an address expression.

For example:

```
(gdb) maint info unwind $pc
modsched:
0x4000930 .. 0x4000a20, end_prologue@0x4000970
Info block version:0x0, flags:0x0, length:4 * 4 == 16
0x40172b20: (0c) R1prologue rlen=12
0x40172b21: (e8) P7preds_when t=11
0x40172b23: (b1) P3preds_gr gr=41
0x40172b25: (ea) P7lc_when t=7
0x40172b27: (b2) P3lc_gr gr=40
0x40172b29: (61) R3body rlen=33
0x40172b2b: (81) B1label_state label=1
0x40172b2c: (c0) B2epilogue t=44
0x40172b2e: (00) R1prologue rlen=0
0x40172b2f: (00) R1prologue rlen=0
```

14.13.4 Printing CFM and PFS registers

On Integrity systems, HP WDB prints Current Frame Marker (CFM) and Previous Frame State (PFS) `ar64` registers in two different formats:

- raw values
- special formats identifying the size of rotating registers, frame and locals.

For example,

```
ar64: 0xc00000000000050c (sor:0, sol:10, sof:12)
cfm: 0x800000000000450a (sor:1, sol:10, sof:10)
```

14.14 Debugging Core Files

14.14.1 Generating core files with `packcore /unpackcore/getcore`

The contents of a core file can be viewed only on a system which has all the shared libraries that were in use on the system on which the core file was generated. If you want to view the content of the core file on a system which does not have the shared libraries, you have to set the environment variables `GDB_SHLIB_PATH` or `GDB_SHLIB_ROOT` to make it search for the desired libraries. The commands `packcore`, `unpackcore`, and `core` simplify the process of examining the contents of a core file on a system other than the one in which it was generated.

The `packcore` command is used on the system which generated the core file. When you are examining the core file on the original system, you can execute `packcore` to make a `packcore.tar.Z file`. This is a compressed tar file which contains the core file, the executable, and all the shared libraries that were used by the program when it generated the core file.The core file is removed after it is added to the `packcore.tar.Z` file.

The `packcore` command has one optional argument, `basename`, which can be used instead of `packcore` to make `packcore.tar.Z`.

The `packcore.tar.Z` file can be copied to a different system and the gdb command `unpackcore` unpacks the `packcore.tar.Z file` in the current directory, creating a new `packcore` directory. After unpacking the `packcore` file, the `unpackcore` command invokes `getcore` to load the executable and the core file from the `packcore` directory, and sets `GDB_SHLIB_PATH` to the `modules` directory in the `packcore` directory. The `modules` directory holds all of the shared libraries that were used when the core file was generated.

The `unpackcore` command has two optional arguments. The first defaults to `packcore.tar.Z` and is the name of the `packcore` file to be unpacked. The second argument is given if the core file is too large to fit in the `packcore` file. It is the path to the core file to be used if the `packcore` directory does not contain a core file. If used,this second argument causes a symbolic link to be created in the `packcore` directory in place of the missing core file.

The `getcore` command can be used to examine a `packcore` directory which was previously created by `unpackcore`. It takes one optional argument, the name of the `packcore` directory, which defaults to `packcore`.

14.14.2 Support for the `dumpcore` command

HP WDB provides the command, `dumpcore` to generate a core image file for a process running under the debugger during execution.

The `dumpcore` command does not require any argument. It saves the core image for the current process being debugged in the file named `core.<pid>`, where `<pid>` is the process ID number.

To dump the core for a live process, you must pass the following commands:

```
(gdb) run  Starting program: sample Breakpoint 3, main () at sample.c:1010 b
= foo(a); (gdb) dumpcore Dumping core to the core file core.24091(gdb)
```

When starting from the HP WDB command line:

```
(gdb) file sample
Reading
symbols from sample...done
(gdb) set live-core 1
(gdb) core-file core.24091
Core was generated by 'sample'.#0 main () at sample.c:1010 b = foo(a);
(gdb) backtrace#0 main () at sample.c:10
(gdb)
```

When starting from the shell prompt:

```
% gdb --lcore a.out core.<pid>
```

For example:

```
% ./gdb --lcore sample core.24091
HP gdb...
Type "show warranty" for warranty/support....
Core was generated by 'sample'.#0 main () at sample.c:10
(gdb)
```

14.14.2.1 Enhancements to the dumpcore command

HP WDB provides an option for the `dumpcore` command, to specify a `<core-filename>`, to generate a core image file of a process running under the debugger in the middle of execution and saves it in the file named `<core-filename>`.

The `dumpcore` command with no arguments saves the core image for the current process being debugged in the file named `core.<pid>`, where `pid` is the process ID number.

To analyze this core file with HP WDB on HP-UX 11i version 2, you must do the following:

When starting from HP WDB command line:

```
(gdb) core-file [core.pid | core-filename]
```

When starting from shell prompt:

```
$ gdb -core a.out [core.pid | core-filename]
```

14.14.3 Support for display of run time type information

HP WDB enables you to view the run time type information for C++ polymorphic object.

`info rtti` *address* This command displays run time type information for C++ polymorphic object. The input to this command is *address* of the C++ polymorphic object. GDB displays the de-mangled type name as output.

Note:

This command is supported only on Integrity systems.

Sample output:

```
(gdb) info rtti <address>
RTTI: <run time type/class name>
```

14.15 Printing the Execution Path Entries for the Current Frame or Thread

HP WDB 5.7 and later versions of the debugger enable you to print the execution path entries in the current frame, or the current thread for programs running on Integrity systems. This feature enables the display of the execution path taken across branched modules. The first instruction in each block associated with the executed branch is displayed.

This feature is supported only for compiler versions A.06.15 and later.

HP WDB supports the following commands to print the execution path entries in the current frame, or in the current thread:

- `info exec-path [start_index] [end_index]` (aliased to `info ep`)

  Lists all the local execution path entries in the current frame. The `[start_index]` and `[end_index]` indicate the range of table indexes (execution path entries) that must be displayed.

  If `[end_index]` is not specified, the debugger displays the complete table of execution path entries, starting from `[start_index]`.

  If `[start_index]` and `[end_index]` are not specified, the complete table of execution path entries is displayed.

  For example:

  ```
  (gdb) i ep 4 10
  ```

- `info exec-path summary`

  Prints the summary information about all the local execution path entries in the current frame. This command displays the total number of branches for the frame, the number of branches executed in this frame in the last iteration, and the last executed branch number.

- `info global-exec-path [start_index] [end_index]`(aliased to `info gep`)

  Lists all the global execution path entries for the current thread.

  The `[start_index]` and `[end_index]` indicate the range of table indexes (execution path entries) that must be displayed.

  If `[end_index]` is not specified, the debugger displays the complete table of execution path entries, starting from `[start_index]`.

  If `[start_index]` and `[end_index]` are not specified, the complete table of execution path entries is displayed.

- `info global-exec-path summary`

  Prints the summary information about all the global execution path entries in the current frame. This command displays the total number of global execution path entries that can be stored, the number of global execution path entries in this frame in the last iteration, and the last executed global execution path number.

- `exec-path [up][down][path_index]` (aliased to `ep`)

  Enables you to select, print, and navigate through the execution path entries. When no arguments are specified, it prints the selected execution path entry. You can specify the argument as an execution path index from the `info exec-path` or the `info global-exec-path` commands. Alternately, you can use the `up` or `down` command to navigate through the execution path entries.

14.15.1 Compiler Dependencies for Printing the Execution Path Entries

The `+pathtrace` compiler option provides a mechanism to record program execution control flow into global path tables, local path tables, or both. This saved information enables the debugger to print the execution path entries in the current thread or frame. To print the execution path entries in the current thread or frame for programs running on Integrity systems, you can set the required sub-options for the `+pathtrace` compiler option.

You must set the following `+pathtrace` compiler option to enable the debugger to print the execution path entries:

`+pathtrace= [<global|global_fixed_size>:<local>]`

For more information on this feature, see the following example.

14.15.2 Example Illustrating Execution Path Recovery

The following example illustrates the use of the execution path recovery feature in HP WDB:

Sample Program:

```
$cat execpath.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
  int a = 3, b = 0, c = 4;

  if (a)
    printf("Value of a greater than 0\n");

  if (b)
    printf("Value of b greater than 0\n");

  if (c)
    printf("Value of c greater than 0\n");

  printf("All condition checking done\n");

  return 0;
}
```

Sample Debugging Session:

```
$cc +pathtrace -g execpath.c
$gdb a.out
HP gdb ...
```

```
Type "show warranty" for warranty/support.
...
(gdb) b main
Breakpoint 1 at 0x4000a60:0: file execpath.c, line 7 from a.out.
(gdb) r
Starting program: a.out

Breakpoint 1, main () at execpath.c:7
7         int a = 3, b = 0, c = 4;
(gdb) n
9         if (a)
(gdb) i ep
Local execution path table for main():
empty
(gdb) i gep
Global execution path table:
empty
(gdb) n
10          printf("Value of a greater than 0\n");
(gdb) n
Value of a greater than 0
12        if (b)
(gdb) i ep
Local execution path table for main():
0       0x4000a80:2             (execpath.c:10)
(gdb) i gep
Global execution path table:
G0      0x4000a80:2     main    (execpath.c:10)
(gdb) n
15        if (c)
(gdb) i ep
Local execution path table for main():
0       0x4000a80:2             (execpath.c:10)
(gdb) i gep
Global execution path table:
G0      0x4000a80:2     main    (execpath.c:10)
(gdb) n
16          printf("Value of c greater than 0\n");
(gdb) n
Value of c greater than 0
18        printf("All condition checking done\n");
(gdb) i ep
Local execution path table for main():
0       0x4000a80:2             (execpath.c:10)
2       0x4000bd0:2             (execpath.c:16)
(gdb) i ep summary
Summary for local execution path table for main()
```

```
Size: 3               \*Total Number of Branch Paths in Current Function
Effective entries: 2   \*Number of Branches executed till this instant
Current entry: 2        \* Last executed branch number
(gdb) i gep
Global execution path table:
G0      0x4000a80:2     main    (execpath.c:10)
G1      0x4000bd0:2     main    (execpath.c:16)
(gdb) i gep summary
Summary for global execution path table
Size: 65536             \*Maximum execution path  entries to be stored
Effective entries: 2     \*Number of global execution path entries
Current entry: 1          \*The last Global Path ID executed
(gdb)
```

14.16 Invoking GDB Before a Program Aborts

This `-crashdebug` option enables GDB to monitor the execution of a process or a program. It invokes GDB when the program execution is about to abort. Once the debugger is invoked, you can debug the application using the normal debugger commands. This option enables you to debug a live process instead of getting a core dump if the program is about to abort.

You can examine the state of the process, make changes to the state and continue program execution, force a core dump, or terminate execution. It enables you to control program execution under the debugger if the program is about to abort. You can load a new process or attach to a running process for monitoring.

To monitor a new process, enter the following command:

```
gdb -crashdebug <command> <options>
```

To monitor a running process, attach to the process using the following command:

```
gdb -crashdebug -pid <pid>
```

14.17 Aborting a Command Line Call

When a command line call is issued and it is interrupted by a breakpoint or a signal before completing the program execution, the `abort` command enables the user to abort the command line call without allowing the signal to modify the state of the debugged process.

When a signal interrupts program execution, it can modify the process state of the debugged program and result in an abrupt termination of the program (due to addressing errors from a call that is not a part of the source program). In such cases, the `abort` command is particularly useful in exiting the command line call without modifying the process state of the debugged program.

The following example illustrates the use of the `abort` command:

```
(gdb) break main
Breakpoint 1 at 0x2c74: file .../address_error.c, line 41.
```

```
(gdb) run
Starting program: ./address_error

Breakpoint 1, main () at ./address_error.c:41
41   fun (count, count*1.1);
(gdb) p fun(10, 1.1)
Program received signal SIGBUS, Bus error
  si_code: 0 - BUS_UNKNOWN - Unknown Error.
0x2c38 in fun (i=10, f=0) at ./address_error.c:37
37   count = *p;
The program being debugged was signaled while in a function called from GDB.
GDB remains in the frame where the signal was received.
To change this behavior use "set unwindonsignal on"
Evaluation of the expression containing the function (fun) will be abandoned.
(gdb) bt
#0  0x2c38 in fun (i=10, f=0) at ../address_error.c:37
#1  0x1920 in _sr4export+0x8 ()
#2  <function called from gdb>
#3  0x2c74 in main () at ./address_error.c:40
(gdb) abort
Abort gdb command line call? (y or n) y
#0  main () at ./address_error.c:41
41   fun (count, count*1.1);
(gdb) bt
#0  main () at ./address_error.c:41
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

14.18 Instruction Level Stepping

During instruction level stepping with `nexti` and `stepi`, WDB prints the assembly instruction along with the next source line.

`(gdb) stepi 0x101530:0 st4 [r9]=r34 1337 args.argc = argc;`

It also prints `DOC` line information, which includes actual line number and the column number, when debugging a binary with `-g -O`.

`(gdb) stepi ;;; [8, 1] 0x4000820:1 nop.m 0x0`

GDB cannot step into a function with no debug information. GDB cannot do a next over a line when there is not debug information. However, the `continue` command works in such situations.

14.19 Enhanced support for watchpoints and breakpoints

14.19.1 Deferred watchpoints

When you try to set a watchpoint in an expression, HP WDB places a deferred watchpoint if HP WDB cannot evaluate the expression. The watchpoint is automatically enabled whenever the expression can be evaluated during the programs execution. This is especially useful when placing the watchpoints on unallocated addresses.

14.19.2 Hardware watchpoints

HP WDB provides support for hardware watchpoints on HP-UX 11.x.

14.19.3 Hardware breakpoints

The `hbreak` command sets hardware assisted breakpoints.

`hbreak args`

The arguments (`args`) is same as that for the break command and the breakpoint is set in the same way. However, the breakpoint uses hardware assisted breakpoint registers. There are only two hardware breakpoints that can be set on Integrity systems. This is useful in ROM code debugging and shared library debugging for libraries, including `dld`, that are not loaded private.

The normal breakpoints are converted to a hardware breakpoint when WDB cannot set a normal breakpoint in the shared library.

14.19.3.1 Setting breakpoints in unstripped shared library

GDB will not be able to put breakpoints using symbolic names(of the symbols not in export list) or line numbers in the stripped modules.

GDB will be able to place breakpoints using symbol names in the unstripped shared libraries loaded into the stripped executable.

14.19.4 Support for procedural breakpoints

HP WDB enables you to set breakpoints at the beginning (first executable line) of every function that can be debugged. In addition, you can specify a set of commands to be performed when the breakpoint is reached. These breakpoints work like procedural breakpoints in the `xdb` debugger.

The breakpoint commands are `rbp` and `rdp`.

- `rbp`: Sets breakpoints at the first executable statement in all the functions that can be debugged, including any shared libraries that are already loaded. The `rbp` command sets breakpoints in all the functions, which can be debugged, in all the source files.After you set these breakpoints, you can manage them like any standard breakpoints. You can delete them, disable them, or make them conditional.Each time you use the `rbp` command, HP WDB adds an additional breakpoint at the beginning of each function that performs the commands you specify, if any.

- `rdp`: Deletes all the breakpoints set by the `rbp` command.

This example shows how to set a breakpoint at the start of each procedure that displays information at the breakpoint:

```
(gdb) file a.out
Reading symbols from a.out...done.
(gdb) rbp
Breakpoints set from 170 to 211
 Type commands to execute when the breakpoint is hit (one command per line).
End with a line saying just "end".
>info break
>end
(gdb)
```

14.19.5 Support for template breakpoints

With HP WDB 5.0, you can set breakpoints on all instantiations of the template class by just specifying the template name with member function name.

For example:

```
(gdb) break ::
```

It is not necessary to specify the instantiation type.

Setting a breakpoint on a template method with multiple instantiations displays a menu showing all instantiations and the user can choose to set breakpoints on all or any one or none.

For example:

```
(gdb) file test
Reading symbols from test...done.
(gdb) b MyClass::MyMember
[0] cancel
[1] all
[2] MyClass::MyMember(int, int) at test.C:14
[3] MyClass::MyMember(int, float) at test.C:14
[4] MyClass::MyMember(int, double) at test.C:14
```

14.20 Debugging support for shared libraries

On HP-UX, shared libraries are special. Until the library is loaded, GDB does not know the names of symbols. However, GDB gives you two ways to set breakpoints in shared libraries:

- deferred breakpoints
- `catch load` command

14.20.1 Using shared library as main program

If the main program is in a shared library and you try to load it as follows:

```
(gdb) symbol-file main.sl
Load new symbol table from "main.sl"? (y or n) y
Reading symbols from main.sl
done.
```

Things don't appear to work.

This command is not the correct thing to do. This command assumes that `main.sl` is loaded at its link time address. This is not not true for shared libraries.

Do not use `symbol-file` with shared libraries.

Instead, what you should do is to use the deferred breakpoint feature to set breakpoints on any functions necessary before the program starts running.

```
(gdb) b main
Breakpoint 1 (deferred) at "main" ("main" was not found).
Breakpoint deferred until a shared library containing "main" is loaded.
(gdb) r
```

Once the program has started running, it will hit the breakpoint. In addition, the debugger will then already know about the sources for `main`, since it gets this information when the shared library is loaded.

14.20.2 Setting Deferred Breakpoints in Shared Library

On HP-UX, GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library—unless you are debugging a core file.)

When you specify a breakpoint using a name that GDB does not recognize, the debugger warns you with a message that it is setting a deferred breakpoint on the name you specified. If any shared library is loaded with a matching name, then GDB sets the breakpoint.

For example, if you type:

'`break foo`'

the debugger does not know whether `foo` is a misspelled name or whether it is the name of a routine that has not yet been loaded from a shared library. The debugger displays a warning message that it is setting a deferred breakpoint on `foo`. If any shared library is loaded that contains a `foo`, then GDB sets the breakpoint.

If this is not what you want (for example, if the name was mistyped), then you can delete the breakpoint.

14.20.3 Using `catch load`

The '`catch load <`*`libname`*`>`' command causes the debugger to stop when the particular library is loaded. This gives you a chance to set breakpoints before routines are executed.

14.20.4 Privately mapping shared libraries

In cases where you attach to a running program and you try to set a breakpoint in a shared library, GDB may generate the following message:

```
The shared libraries were not privately mapped; setting a breakpoint
in a shared library will not work until you rerun the program.
```

GDB generates this message because the debugger sets breakpoints by replacing an instruction with a `BREAK` instruction. The debugger cannot set a breakpoint in a shared library because doing so can affect other processes on the system in addition to the process being debugged.

To set the breakpoint you must kill the program and then rerun it so that the dynamic linker maps a copy of the shared library privately. There are two ways to run the program:

- Rerun the program under GDB to have the debugger cause `dld` to map all shared libraries as private, enabling breakpoint debugging.
- On PA-RISC systems, use the following command on an executable:

  '`/opt/langtools/bin/pxdb -s on` *executable-name*'

  The `pxdb -s on` command marks the executable so that `dld` maps shared libraries as private when the program starts up.

  You can verify the status of the shared library with this command:

  '`/opt/langtools/bin/pxdb -s status` *executable-name*'
- On both PA-RISC and IA64 systems, use the following command on an executable:

  '`chatr +dbg enable` *executable-name*'

  This is similar to the `pxdb` command described above wherein it directs the `dld` to load the shared libraries as private when the program starts up.

14.20.5 Selectively Mapping Shared Libraries As Private

The `-mapshared` option suppresses mapping all shared libraries in a process private. This option enables new functions in the dynamic loader (`patch PHSSS_33110` or later) to designate individual shared libraries for debugging. By default, HP WDB instructs the shared library dynamic loader, `dld.sl(5)`, to map shared libraries in a process private, regardless of whether the `chatr` command is run for a particular `shlib` with `+dbg` or not.

The `-mapshared` option is used to save virtual memory for debugging applications with large amounts of code in shared libraries on machines with simultaneous debug sessions. The `chatr +dbg` option, and the `_HP_DLDOPTS` environment variable are used to identify shared libraries for debugging. The `-mapshared` option ensures that the text segments of all other shared libraries is shared across the system. The shared libraries are not mapped private and cannot have breakpoints set in them.

The `set mapshared on` command can be used to change modes from the `(gdb)` prompt.

```
(gdb) set mapshared on
```

The `set mapshared off` command can be used to load shared libraries after the current point is mapped private.

```
(gdb) set mapshared off
```

The `-mapshared` option is implemented on both PA-RISC and Itanium platforms in HP WDB 5.2. This option is provided in the WDB GUI and HP WDB. The default behavior does not change if the `-mapshared` option for all shared libraries in processes started under the debugger, or dynamically loaded after an attach, are mapped private.

14.20.6 Setting breakpoints in shared library

Breakpoints can be set on functions in a shared library, by specifying the library name with the function name.

```
(gdb) b libutil.sl:fun          Breakpoint 1 at 0x79a86228: file simple.c,
line 13 from /CLO/Components/WDB/build/hppa1.1-hp-hpux11.00/gdb/
simple_shared/lib.sl
```

14.21 Language support

14.21.1 Enhanced Java Debugging Support

The following commands have been added to enhance Java debugging support:

`backtrace`
: Prints `backtrace` of mixed Java and native frames. Standard `backtrace` command of GDB has been enhanced to work with mixed Java and native stack frames.

`info frame`
: Prints Java frame specific information for a Java frame. Standard frame command of GDB has been enhanced to interpret a Java stack frame.

`info threads`
: Prints state information for Java threads.

`thread`
: Prints detailed state information for the current Java thread.

14.21.2 Commands for Examining Java Virtual Machine(JVM) internals

The following commands can be used to obtain information on Java VM internals:

`help java`
: Prints list of Java and JVM debugging commands

`java`
: Prints list of java subcommands

14.21.2.1 Java subcommands

The following Java subcommands have been added:

`java args <frame-number>`
: Prints the current or specified Java frame arguments information

`java bytecodes <methodOop>`
: Disassembles the given Java method bytecode

`java heap-histogram`
: Displays the Java heap object histogram

`java instances <klassOop>`
: Locates the instances of the given klassOop in the Java heap.

`java jvm-state`
: Prints current status of JVM internal states

`java locals`
: Prints the current or specified Java frame locals information

`java mutex-info`
: Prints details of static mutexes

`java object <object-ptr>`
: Prints the given Java object field information

`java oop <Java_heap_address>`
: Locates Java object oop of the given Java heap address

`java references <oop>`
: Locates references to the given Java object in the Java heap

`java unwind-info <pc>`
: Prints unwind information of the code where the PC is located

`java unwind-table`
: Prints the dynamically generated Java Unwind Table

14.21.3 Support for stack traces in Java, C, and C++ programs

HP WDB shows stack traces of mixed Java, C, and C++ programs. The stack trace functionality requires Java SDK version 1.3.1.02 or later for HP-UX. To find the availability of Java SDK version 1.3.1.02 or later, go to the HP web site for Java, http://www.hp.com/go/java.

To enable this feature, set the environment variable, `GDB_JAVA_UNWINDLIB`, to the path to `libjunwind.sl`. This library is part of the Java SDK version 1.3.1.02 or later for HP-UX. When `GDB_JAVA_UNWINDLIB` is set to the path for a valid Java unwind library, stack traces display Java and C/C++ frames.

The types of Java frames supported are as follows:

- Interpreter
- Compiled frames
- Adapter frames

14.21.4 Support for 64-bit Java, C, aC++ stack unwinding

This release of HP WDB can show stack traces of mixed Java, C, and C++ programs in the 64-bit mode. The stack trace functionality requires Java SDK version 1.4 or later for HP-UX. Please check the HP web site for Java, http://www.hp.com/go/ java, for availability of Java SDK version 1.4 or later.

To enable this feature, set the environment variable GDB_JAVA_UNWINDLIB to the path to a libjunwind.sl. This library is part of the Java SDK version 1.3.1.02 or later for HP-UX. When GDB_JAVA_UNWINDLIB is set to the path for a valid Java unwind library, stack traces will show Java and C/C++ frames.

In this version, even if GDB_JAVA_UNWINDLIB is not set, HP WDB uses the libjunwind.sl specified by the Java Virtual Machine.

The types of Java frames supported are as follows:

- Interpreter
- Compiled frames
- Adapter frames

14.21.5 Enhanced support for C++ templates

This version of HP WDB includes these features to support C++ templates:

- Setting breakpoints in template class functions and template functions without having to specify details about the instantiation.
- The `ptype` command shows any one of the class instantiations.

A option `-v` in ptype command will now display the field offset and size information of a `struct/union/class` in addition to the default type information.

Syntax:

`ptype [-v] [struct|union|enum|class]`

Example:

```
(gdb) ptype -v struct info
type = struct info  /* off 0 bits, len 512 bits */
    int i;
        /* off 0 bits, len 32 bits */
    char a[20];
        /* off 32 bits, len 160 bits */
    struct details d;
        /* off 192 bits, len 256 bits */
    int b : 2;
        /* off 448 bits, len 2 bits */
    int c : 3;
        /* off 450 bits, len 3 bits */
    < filler >
        /* off 453 bits, len 27 bits */
    float f;
```

```
                /* off 480 bits, len 32 bits */
```

14.21.6 Support for `__fpreg` data type on IPF

WDB internally converts `__fpreg` data type to long double data type to evaluate an expression or to print the value of the expression. Since long double data type has only 15 bit exponent as opposed to 17 bit exponent of `__fpreg`, some precision is lost when the exponent is larger than that can fit in 15 bits.

14.21.7 Support for _Complex variables in HP C

HP C on Itanium systems supports a `_Complex` data type built from any of the floating point types.

A `_Complex` number holds a pair of floating point numbers; the first is the "real part" and the second is the " imaginary part".

Here are examples of declarations and initializations using `_Complex` numbers:

```
float _Complex glob_float_complex;
double _Complex glob_double_complex = 6;
long double _Complex glob_long_double_complex = _Imaginary_I;
__float80 _Complex glob_float80_complex = 8 + 9 * _Imaginary_I;

_Imaginary_I is a keyword which represents the square root of -1.
```

The debugger has limited support for `_Complex` variables. No arithmetic operations are allowed with `_Complex` numbers. A `_Complex` number may be cast or assigned to any numeric data type and vice versa.

A `_Complex` variable can be initialized with an expression of the form:

```
A + B * _Imaginary_I
```

Where `A` and `B` are ordinary numeric expressions, perhaps in parentheses.

This is also the format in which the debugger displays a _Complex value.

Imaginary values cannot be assigned to variables because there is no imaginary data type. You can take a normal number and multiply it by an imaginary number and get another imaginary number. You can take a normal number and add it to an imaginary number to get a complex number.

Complex numbers cannot be used in arithmetic expressions in the debugger.

For more information of `_Complex` type, refer to the HP C/ANSI C documentation.

14.21.8 Support for debugging namespaces

This release of HP WDB provides full support for debugging namespaces.

You do not need to use fully qualified names to access symbols within a namespace. The debugger can compile a list of namespaces active in the scope that you are in and, when possible, choose an appropriate symbol.

The debugger recognizes using declarations, using directives, namespace aliases, nested namespaces, and unqualified lookup within a namespace. It also recognizes using directives and aliases, and using declarations within namespaces.

When the debugger has a list of possible resolutions for a given symbol, it displays a menu that shows all names fully qualified whenever namespaces are involved. You can choose the appropriate symbol from the list.

For example, if you stop the debugger in a function that contains an `int i` using directive for a namespace such as:

```
using namespace A::AB::ABC::ABCD
```

You can use the command `print i` and if the only possible resolution for i is A::AB::ABC::ABCD::i the debugger prints out the name of the symbol and its value. If, however, a global i exists, the debugger displays a menu from which to choose:

```
(1) i
(2) A::AB::ABC::ABCD::i
>
```

Setting breakpoints on functions works in the same way.

The debugger also allows semi-qualified names. For example, if you stop in a function in namespace B, which is nested in namespace A, and namespace A has an `int i`, you can use print B::i to display the value of A::B::i.

To disable namespace support, use the command:

```
(gdb) set namespaces-enabled off
```

14.21.9 Command for evaluating the address of an expression

The `watch_target` command takes an expression as an argument, evaluates it to an address, and watches the target of that address.

For example:

```
(gdb) watch_target current_frame
```

This is equivalent to executing:

```
(gdb) print current_frame  $1 = (struct frame_info *) 0x7fdf78  (gdb) watch
*(struct frame_info *) 0x7fdf78
```

14.22 Viewing Wide Character Strings

HP WDB `print` command can print wide characters and wide-character strings of the type `wchar_t`. The user must use the `/W` option of the `print` command to print wide characters and wide-character strings.

```
print /W <wide-char-symbol-name>
```

14.23 Support for output logging

The Visual Interface for HP WDB terminal user interface (TUI) mode supports the command, `log logfile_name`, that saves the content of a session to the specified log file.

When you use the `log` command, the debugger saves a snapshot of the current session, from the start of the session to the point where you issued the `log` command. Each time you use the `log` command, HP WDB overwrites the specified log file with a new snapshot from the start of the session.

To run the Visual Interface for HP WDB, use the following command:

```
$vdb -tui
```

To redirect HP WDB output to a log file named `mylogfile`, use the `log` command in the following manner:

```
(gdb) log mylogfile
```

The Visual Interface for HP WDB stores the log file, `mylogfile`, in the current directory.

To view the log file from Visual Interface for HP WDB, start a shell process and use the following command:

```
(gdb) shell vi mylogfile
```

14.23.1 Support for dumping array in an ASCII file

HP WDBN supports dumping an array into an ASCII file.

The array elements are stored in Array format of Matrix Market in a predefined (column-major order for Fortran arrays) order. The objective is to provide a simple mechanism to facilitate the exchange of matrix data and to enable easier parsing of the array elements.

For common file formats, see http://math.nist.gov/MatrixMarket/formats.html.

To dump an array, `ARRAY`, to a file named `DUMPFILE`, use the following command:

```
(gdb) dump2file ARRAY DUMPFILE
```

The entries of ARRAY are dumped into an ASCII file named `DUMPFILE` in the array format. The file is created in the current working directory. The content of the file has the following format:

```
%%ArrayBrowsing matrix array ARRAY
% A 5x5 matrix
5 5
0
2
4
6
8
2
..
..
```

where, `ARRAY` is the name of the array, and its size is 5x5.

The first two lines are comments about this file and the array. The third line denotes the array coordinates. From the fourth line, the elements of the array are listed.

Note: This feature is not supported for the Fortran array slices.

14.23.2 Support for Fortran array slices

HP WDB prints Fortran array slices if you specify a range of elements by the Fortran 90 array section syntax. For instance, for an `array X` declared by `REAL, DIMENSION(-1:1, 2:10) :: X`, you could print all five even-numbered elements of the row with the first dimension equal to 0 by typing the WDB command `print X(0,2:10:2)`.

14.23.3 Displaying enumerators

You can display the union of several enumeration elements by specifying a value if the elements of the enumeration type are the powers of 2 and the given value is a sum of any given combination of the enumeration elements.

For example, assume you have an enumerated type named color in the program, with these elements: `RED=0, ORANGE=1, YELLOW=2, GREEN=8, and BLUE=16`. If you use the command `printf 3`, the debugger displays `ORANGE|YELLOW`, the elements corresponding to 1 and 2. If you print 5, you will get the value, 5, because it does not form the sum of any combination in the set. However, if you wanted to print 25, you will get Orange|Green|Blue.

Values that do not form the sum of any combination of the elements will be displayed as integers while the values that form the sum of any combination of the elements will be printed as unions.

14.23.4 Support for debugging typedefs

When you have a `typedef` class as a template parameter, you can set a breakpoint on a member function by using the command:

```
break Class<typedef_classB>::memfunc
```

14.23.5 Support for `steplast` command for C and C++

Typically, if a function call has arguments that make further function calls, executing a simple `step` command in GDB steps into the argument evaluation call. HP WDB includes the `steplast` command, which helps to step into a function, and not into the calls for evaluating the arguments. However, the `steplast` command is not available on Integrity systems. The following example illustrates how GDB behaves when you execute the `steplast` command:

`(gdb) 16 foo (bar ()); ---> bar() will return 10 (gdb) steplast foo (x=10) at foo.c:4 4 int k = 10;`

If the `steplast` command is not meaningful for the current line, GDB displays the following error message:

`"Steplast is not meaningful for the current line."`

For example,

`(gdb) 4 int k = 10; (gdb) sl ---> alias to "steplast" command error: Steplast is not meaningful for the current line`

To execute the `steplast` command in C++ compiled applications, you must compile the application using the HP aC++ version A.03.50 or later with the `-g0` option.

In C++, the `steplast` command is helpful while debugging heavy templated functions, because it directly steps into the call, thus skipping the constructor calls, if any. This behavior is unlike the `step` command that steps into the constructor itself.

Consider the following example:

```
void call_me ( string s )  ... (gdb)
10
call_me ( "hello" );
(gdb) steplast call_me (s=static npos = 4294967295,
static nullref = ref_hdr = mutex_= dummy1 = 0x7f4f79e0, dummy2 = 2136325568,
refs_ = 2136327612,
capacity_ = 2136327468, nchars_ = 2136327464, eos_char = 64 '@',
alloc_ = <No data fields>,
value_allocator = alloc_ = 0x7f7f133c,
data_ = 0x40003a64 "hello") at str.C:55
printf ("Will just print the value of \n");
```

If there are multiple top-level calls, the `steplast` command enables you to step into each top-level call. For example, for the following line, the `steplast` command takes you to the first top-level call, `(foo())`:

`foo(bar()) + bar(foo());`

Debug `foo()`, use the `finish` command to exit from the first top-level call, `(foo())`, execute the `steplast` command to step into the next top-level call, `(bar())`. The following example illustrates the use of `steplast` command:

`(gdb)10 foo( bar() ) + bar( foo() ) (gdb) sl Use the steplast (sl) command to step`

14.24 Getting information from a non-debug executable

You can get some information about the arguments passed to the functions displayed in the stack trace in a non-debug, optimized executable.

When GDB has no debug information; it does not know where the arguments are located or even the type of the arguments. GDB cannot infer this in an optimized, non-debug executable.

However, for integer arguments you can find the first few parameters for the top-of-stack frame by looking at the registers. On PA-RISC systems, the first parameter will be in `$r26`, the second in `$r25` and so on. On IPF systems, the first few parameters will be in `$gr32` and `$gr33`.

14.25 Debugging optimized code

HP WDB supports debugging of optimized code (compiled with both `-g` and `-O`) for HP aC++, HP ANSI C and HP WDB for HP Itanium.

The following commands evaluate the name of a function and hence are affected by the optimization level of the program being debugged (in particular, due to inlining):

- `break`
- `call`
- `clear`
- `disassem`
- `list`

The following commands evaluate an expression referring to variables in the user program and hence, are affected by the optimization level of the program being debugged:

- `break`
- `call`
- `cond`
- `jump`
- `return`
- `print`
- `set <var>`
- `watch`
- `whatis x`

Note: The `break` and `call` commands involve evaluation of both the name of a function and an expression.

The following commands are also affected by the optimization level of the program being debugged:

- `backtrace`
- `display`
- `down`
- `finish`
- `frame`
- `info *`
- `next`
- `step`
- `tbreak`
- `rbreak`
- `up`

The following commands are not affected by the optimization level of the program being debugged:

- `attach`
- `catch`
- `commands`
- `continue`

- `core`
- `delete`
- `define`
- `detach`
- `disable`
- `enable *`
- `exec`
- `file`
- `forw`
- `handle *`
- `help *`
- `ignore`
- `kill`
- `load`
- `nexti`
- `path`
- `quit`
- `rev`
- `run`
- `set args, set env, set <param>`
- `show args, show <param>`
- `signal`
- `source`
- `stepi`
- `symbol`
- `target`
- `tty`
- `undisplay`
- `unset env`
- `until`

14.25.1 Debugging Optimized Code at Various Optimization Levels

The following sections describe debugging optimized code support at each optimization level.

14.25.1.1 `+O0` and `+O1`

At `+O1` level, optimizations that affect the user visible state of a program are avoided. Line numbers are accurately associated with each machine instruction. Global or local variables may be examined, except for unused variables (which may be eliminated). New values may be assigned to a global and a local variable (`set <var> = <expression>`) when stepping by line (`step/next/break <line>`). However, while stepping by instruction (`stepi/nexti`) at optimization level `+O1`, assign a value to a variable only if stopped at the very first instruction. This is a must as local optimizations are performed within a statement.

Backtrace commands `(backtrace)` may be used to display the current nest of function calls, including for calls that are inlined. Note that even at `+O1`, C++ methods that are defined within a class and Fortran arithmetic statement functions are implicitly inlinable and are inlined. Other functions are not inlined, regardless of the inline pragmas or keywords.

14.25.1.2 `+O2/+O3/+O4/-ipo`

Stepping by line number (`step/next`) and running to a breakpoint(`break`) moves the state of a program forward. However, the program execution does not necessarily stop at the given line.

You can set breakpoints (`break`) at the entry to a routine that is not inlined and examine the values of parameters when the program execution stops at the entry of a routine. The local variables can be examined within a function. However, the values of the local variables may not be available at all code locations in the function. Assignment of new values to formal parameters or local variables is NOT supported in code compiled with optimization above `+O1`.

Optimization of code results in the reordering of the instructions and the source line-numbers. Hence, the value of the variable, which is printed by the debugger may not correspond to the reported source code location. The debugger may print the value of the variable at a source code location either before or after the reported source code location.(If the printed value is not current with respect to the current source line, the printed value will be the immediately previous or immediately later value for the variable.)

Backtrace commands (`backtrace`) can be used to display the current nest of function calls, including calls that are inlined. When stopped within the code for an inlined call, the parameters and the local variables of the inlined routine are not reported or available. The `disassem` command does not work for functions that have no code (because all calls to these functions are inlined or these functions are not called at all).

HP WDB 5.7 and later versions provide support to prevent the debugged program from stopping at instructions that are predicated false. The program execution can be stopped by a software breakpoint, a hardware breakpoint, or an asynchronous signal. In the case of optimizations such as if-conversion, the predicated false instructions indicate that an alternate source path is executed. Hence, stopping the program at a predicated false instruction results in the misleading conclusion that the path corresponding to the predicated false instruction is executed. To prevent this ambiguity, HP WDB does not stop at predicated false instructions.

The predicated false instructions are equated to `NOPs` (No OPeration), because these instructions do not modify the processor state. The exception to this rule is the use of certain instructions, such as `wtop`, `wexit`, and `frcpa`, which modify the processor state even when predicated false. In such cases, the debugger stops at the instructions irrespective of the predicate value of the instructions. Assembly and low-level programmers, who require the old behavior of the debugger to stop at the instructions irrespective of the predicate value of these instructions, can explicitly turn off this feature. To explicitly turn off this feature, enter the following command at the gdb prompt:

```
(gdb) set no-predication-handling
```

The following limitations apply when debugging optimized code:

- Support for high-level loop transformations such as modulo-scheduled loops, or LNO-optimized loop nests is limited. (This limited support includes all loop optimizations that are enabled at `+O3` and above, and some loop optimizations at `+O2` or `-O`.)
- Debug support for local aggregates and arrays is limited.
- Complete debug support for inlined subroutines is not available.
- Values that are not at the current code location will be reported as being unavailable, even if these values can be computed from some other values that are available.
- Step operations may include occasional "backwards" steps, because of the reordered code during optimization.
- The program stops at asynchronous signal stops even if the reported instruction is predicated false.

Complete support is available for debugging at the assembly language level. Stepping by instructions (`stepi/nexti`) steps as expected and reports the associated source line numbers for each instruction.

> **Note:** The `-ipo` compilation implies the `+noobjdebug` option because the `-ipo` object files do not store executable code or debug info.

14.26 Visual Interface for WDB

WDB includes an HP-supported Visual Interface for WDB with both graphical and terminal modes. The interface is based on Vim 5.7 and WDB. This interface replaces the `-tui` mode on Itanium-based systems.

When you use the interface you are actually using `vim`, which is a `vi`-compatible editor. With the interface you can use `vi` commands to browse in the WDB display.

Most of Visual Interface for WDB functionality is also available for emacs users. Visual Interface for WDB does not require knowledge of `vi` commands.

Visual Interface for WDB identifies you as an emacs user by looking at the environment variable '`$EDITOR`'. If this variable has a value that matches emacs, or gmacs, or xemacs, then Visual Interface for WDB starts in emacs mode automatically.

> *Note:* If the program expects unbuffered input or uses curses, termcap, or terminfo, or otherwise transmits escape or control sequences to the terminal, you must use one of the following methods to run Visual Interface for WDB:

- Start the process in one terminal and attach to it with Visual Interface for WDB.
- Use the '`tty`' command at the debugger prompt so the program's input and output are directed to another terminal.

Note: if the underlying GDB terminates abnormally when you are using Visual Interface for WDB, do not close the Visual Interface for WDB window. Wait for a minute or two. Visual Interface for WDB captures the stack trace and the debugging session details and sends you an email. You can then forward this to HP when you report the problem. This is helpful to HP in reconstructing the crash scenario.

14.26.1 Starting and stopping Visual Interface for WDB

You can use Visual Interface for WDB in either of two modes:

- X-window-based graphical interface: Supports mouse and keyboard commands.
- Terminal interface: Supports keyboard commands only.

Visual Interface for WDB accepts the same command line arguments as GDB so you can add options to the startup command. See the man page for GDB for the list of arguments.

- To start Visual Interface for WDB in graphical mode with mouse support, run Visual Interface for WDB with the command:
  ```
  /opt/langtools/bin/vdb
  ```
- To start Visual Interface for WDB in terminal user interface mode, run Visual Interface for WDB with the command:
  ```
  /opt/langtools/bin/vdb -tui
  ```
- To stop Visual Interface for WDB, type `quit` on the WDB command line:
  ```
  (wdb) quit
  ```

14.26.2 Navigating the Visual Interface for WDB display

The Visual Interface for WDB window consists of two areas:

- Source pane at the top
- Debugger pane at the bottom

You can use the arrow and pagination keys on the keyboard to move the cursor:

- Pagination keys move the cursor in the source window, at the top, above the status line.
- Holding the shift key down while using the pagination keys moves the cursor in the debugger window.
- The up and down arrow keys move the cursor in the source window.
- Holding the shift key down while using the up and down arrow keys move the cursor in the debugger window.
- The left and right arrow keys move the cursor in the debugger window.

- Two rows of labeled softkeys at the bottom of the display give you quick access to common commands.

Visual Interface for WDB GUI display

```
     33 #include <stdlib.h>
     34 #include "Deck.h"
     35 #include "Player.h"
     36 #include "House.h"
     37
     38 int main ()
     39 {
*>   40    srand ((int) time(0));
     41
     42    Deck theDeck;
     43    Player thePlayer (100);
     44    House theHouse (16);
     45
     46    theHouse.Instructions();
     47  }
File: BlackJack.C    Function: main    Line: 40    Pc: 0x3ea0
(wdb) b main
Breakpoint 1 at 0x3ea0: file BlackJack.C, line 40.
(wdb) run
Starting program: /work/wdb/blackjack/blackjack

Breakpoint 1, main () at BlackJack.C:40
40         srand ((int) time(0));
(wdb)
```

Run	Resume	Stop	Up	Visual Interface for WDB	Finish	Print	Type	List
Faq	Stop	Next	Down		Prompt	Print*	Edit	Credits

You can click the softkey or press a function key on the keyboard to invoke the command.

The function keys F1 through F8 correspond to the bottom row of softkeys. The function keys F9 and up correspond to the top row.

14.26.3 Specifying foreground and background colors

To change the foreground and background colors, update the '`.Xdefaults`' file in the home directory. The resources are the same as for '`hpterm`'.

Here is a sample entry:

```
HPterm*foreground:          white
HPterm*background:          rgb:68/90/C4
```

14.26.4 Using the X-window graphical interface

To start Visual Interface for WDB in graphical mode with mouse support, run Visual Interface for WDB with the command:

```
/opt/langtools/bin/vdb
```

Visual Interface for WDB opens an '`hpterm`' window, ignoring the value of the `TERM` environment variable, for debugging a program.

With a mouse you can do the following:

- Left-click the line number to insert or remove breakpoints.
- Left-click an identifier to select the identifier as an operand for the Print, Print*, Type, and List softkeys.
- Where necessary, manually select an expression by dragging the cursor over it.
- Right-click the line number to activate a pop-up menu with several useful commands.
- Right-click an identifier to automatically select it and use the selection as an operand for the pop-up window that appears.
- Right-click an empty region for a third pop-up menu with several useful actions. For example, see Section 14.26.11 [Saving session to file], page 193.
- Left-click the command softkeys at the bottom of Visual Interface for WDB window.
- Click the middle button to paste the selection.
- Drag the status bar with the mouse to resize the debugger window relative to the source window.

14.26.5 Using the TUI mode

To start Visual Interface for WDB in terminal user interface (TUI) mode, run Visual Interface for WDB with the command:

```
/opt/langtools/bin/vdb -tui
```

This mode works well with `hpterm` and `xterm` and fairly well with `dtterm` and VT100 (telnet) terminals.

Note: A defect in `dtterm` may truncate the display of lines that do not fit within the window. To work around this defect, refresh the display with ⟨CTRL⟩-L or widen the terminal window so source lines do not wrap.

If you use `xterm` and `dtterm`, update the '`.Xdefaults`' file with keyboard translations to get the shifted arrows and shifted paging keys to work.

For `xterm`, use the following:

```
          XTerm*vt100.translations: #override \
          Shift <Key>Prior:    string(0x2)  \n \
          Shift <Key>Next:     string(0x6)  \n \
          Shift <Key>Up:       string(0x5)  \n \
          Shift <Key>Down:     string(0x19) \n \
          Shift <Key>Left:     string(0x1b) string(i) \n \
          Shift <Key>Right:    string(0x1b) string(la)
        For DtTerm use the following:
          *DtTerm*Translations: #override\n \
          Shift <Key>osfPageUp: string(0x2)  \n \
          Shift <Key>osfPageDown: string(0x6)  \n \
          Shift <Key>osfUp:       string(0x5)  \n \
          Shift <Key>osfDown:     string(0x19) \n \
          Shift <Key>osfLeft:     string(0x1b) string(i) \n \
          Shift <Key>osfRight:    string(0x1b) string(la)
```

Mouse operations are not supported in the `-tui` mode. Also the paging and shift keys do not work well with VT100.

14.26.6 Changing the size of the source or debugger pane

1. Escape to vi command mode first.
2. Drag the status bar using the mouse.

If you are using `-tui` mode, use these commands to change the size of the current window:

⟨CTRL⟩`-W +`
: to increase

⟨CTRL⟩`-W -`
: to decrease

In Visual Interface for WDB, the current window is usually the debugger window.

14.26.7 Using commands to browse through source files

Visual Interface for WDB is based on '`vim`', so you can also use the '`vi`' commands to browse. For example, ⟨CTRL⟩-B, ⟨CTRL⟩-F, ⟨CTRL⟩-D, ⟨CTRL⟩-U are useful for browsing the debugger window. These commands work whether or not you escape to '`vi`' mode.

These '`vim`' commands require you to escape to '`vi`' mode.

For example:

'`/`'	Search forward
'`?`'	Search backward

'`n`' '`N`'	Repeat search
'`%`'	Match braces
'`[[`' '`]]`'	Skip to the next procedure
'`:`*`line number`*'	Go to any line number

All these commands require you to escape to '`vi`' command mode first. When you are done, type `a` for append or `i` for insert or other '`vi`' commands to return to text insertion mode.

Or you can simply click the Prompt softkey.

14.26.8 Loading source files

Escape to '`vi`' command mode and use the `:e` command to load a source file.

```
:e filename
```

When the source files are located in multiple directories, you can simply specify the base name alone as long as file names are unique and the appropriate `dir` commands have been executed.

Pressing the Prompt softkey takes you to the command prompt and also updates the source window so that the cursor remains where the program is stopped.

14.26.9 Editing source files

To edit a file, kill the process then click the `Edit` button. If you do not kill the process, the source file and binaries can get out of sync.

14.26.10 Editing the command line and command-line history

Visual Interface for WDB preserves the entire session's transactions so you can browse through these at any time.

To edit the command line, press ⟨ESC⟩ to enter `vi` mode and then use `vi` commands. You can recall previous commands in history by using `[jk^P^N]`. Complete command lines using ⟨TAB⟩.

14.26.11 Saving the contents of a debugging session to a file

You can save the contents of the current debugging session, including debugger input/output and program input/output) to a file.

To save a session to a file:

1. Right-click an empty region of the source or debugger pane.

2. Choose "Save Session to vdb.*pid*" from the pop-up menu.

   The debugger writes the input and output to a file whose name ends in the `pid` of the debugger. If you save the session more than once, the new transactions are appended to the file.

14.27 Support for ddd

GDB works with `ddd`, the free GDB GUI shell available at `http://mumm.ibr.cs.tu-bs.de/`. While this is not formally supported by Hewlett-Packard, these two do work together. Note however if you have `ddd` issues, you'll need to report them to the `ddd` support channel.

14.28 Support for XDB commands

HP WDB provides support for a subset of XDB commands, enabled with the `-xdb` option.

14.28.1 stop in/at dbx commands

The commands `<stop in function/address>` and `<stop at line>` are equivalent of dbx `<break function/address /line>` command. WDB supports the `<stop in/at>` command in non-dbx mode.

For example:

```
$ gdb a.out  (gdb) stop in main  Breakpoint 1 at 0x2a34: file list.c, line 18
from /tmp/a.out  (gdb) stop at 25  Breakpoint 2 at 0x2a6c: file list.c, line
25 from /tmp/a.out (gdb)
```

14.29 GNU GDB Logging Commands

The following commands control GDB logging activities:

- `set logging file`: Set the current log file
- `set logging off`: Set logging off
- `set logging on`: Set logging on
- `set logging overwrite[on|off]`: Set whether logging overwrites or appends to the log file.
- `set logging redirect [on|off]`: Set the logging output mode

14.30 Support for command line calls in a stripped executable

HP WDB enables you to perform command line calls in a stripped executable.

14.30.1 Support for command line calls in a stripped executable on PA-RISC systems

In WDB, to perform command line calls in a shared library without the help of dynamic linker (using `end.o`), you must execute the following command:

```
chatr -B immediate <executable>
```

In addition, modify all the calls to `shl_load()` to specify `BIND_IMMEDIATE`.

To perform command line calls after attaching a running process to GDB, you must execute one of the following commands:

- `/opt/langtools/bin/pxdb -s on <executable>`
- `chatr +dbg enable <executable>`

14.30.2 Additional support for command line calls in a stripped executable

HP WDB enables you to perform command line calls in a stripped executable. The various scenarios in which you can make command line calls in a stripped executable are as follows:

14.30.2.1 For 32-bit applications:

To perform command line calls in a shared library, without the help of dynamic linker (using `end.o`), you must perform the following operations:

- Execute the `chatr -B immediate <executable>` command.
- Modify all the calls to `shl_load()` to specify `BIND_IMMEDIATE`, if any.

To perform command line calls after attaching GDB to a running process, without the help of dynamic linker (using `end.o`), you must do the following for the program:

- Execute the `chatr -B immediate <executable>` command
- Modify all the calls to `shl_load()` to specify `BIND_IMMEDIATE`, if any.
- Execute the `/opt/langtools/bin/pxdb -s on <executable>` or `chatr +dbg enable <executable>` command.

  To avoid changing of the run-time binding behavior of a program to `BIND_IMMEDIATE`, to perform command line call, do the following:

  - Use the linker option, `+ea`, to export symbols from an object file.
  - Install the linker patch, PHSS_28870 (for 11.0) or PHSS_28871 (for 11i).
  - Execute the following commands:

    ```
    cc -c file.c
    cc file.o -Wl,+ea,/opt/langtools/lib/end.o -s
    ```

14.30.2.2 For 64-bit applications

To perform command line calls in a stripped executable, linked with `end.o`, you need to do the following:

- In the `+std` link mode, GDB supports this feature without any changes. You must export the `__wdb_call_dummy` symbol as shown in the next line.
- In the `+compat` link mode, execute the following command:

```
cc +DD64-g file.c -Wl,+ee,__wdb_call_dummy -s
```

14.30.3 Support for debugging stripped binaries

HP WDB provides limited support for debugging stripped binaries.

14.30.3.1 Printing of locals and globals in a stripped module

GDB will not be able to print the locals and statics declared in a module which has been stripped. GDB will be able to print the exported symbols since exported symbols are not stripped with strip command (they stay in `.dynsym`).

GDB will be able to access the globals or locals defined in other unstripped shared libraries loaded into the stripped executable when you are in the right scope.

14.30.3.2 Backtrace on stripped frames

GDB should be able to backtrace properly stripped frames. Arguments will not be displayed (as in the case of non `-g` binary). If it is a fully archived stripped binary, function names will not be displayed (but PCs will be).

14.30.3.3 Command line calls to non-stripped library

Command line calls to the functions (exported symbols) in the stripped binary work fine. Command line calls to the non-stripped library work normally regardless where the process is stopped.

14.30.3.4 Setting breakpoints in unstripped shared library

GDB will not be able to put breakpoints using symbolic names(of the symbols not in export list) or line numbers in the stripped modules.

GDB will be able to place breakpoints using symbol names in the unstripped shared libraries loaded into the stripped executable.

14.31 Displaying the current block scope information

The `which` command takes a symbol as an argument and prints the information on a given symbol. It prints the following information:

- current block scope addresses
- line information of the definition of the symbol
- filename in which the definition of the symbol occurs

The `which` command does not work for global and type symbols since they do not contain line information.

Syntax:

`which <symbol>`

For example : `(gdb) which i Line 4 of "example.c" block starts at address 0x29a8 <main> and ends at 0x29e4 <main+0x3c>`

14.32 Linux support

Linux Runtime Environment (LRE) on HP-UX Itanium enables users to execute Intel Itanium Linux applications on HP-UX. HP WDB provides a prototype for LRE debugging, which allows you to debug applications ported from Linux that run under LRE. This provides a minimal debugging capability for LRE.

15 The HP-UX Terminal User Interface

By default, GDB runs in line mode. For users who prefer an interface similar (though not identical) to that of the XDB debugger, HP provides a terminal user interface (TUI), which appears when you invoke the `gdb` command with the `-tui` option.

Use the `-xdb` option to enable the use of a number of XDB commands. See the Chapter 16 [XDB to WDB Transition Guide], page 213.

15.1 Starting the TUI

Invoke the debugger using a command like the following:

```
gdb -xdb -tui a.out
```

These examples use the default terminal screen size of 24 by 80 characters. Figure 1 displays the terminal screen window:

Figure 1: The terminal window

```
    |----------------------------------------------------------------------|
    |30       {                                                            |
    |31           /* Try two test cases. */                                |
    |32           print_average (my_list, first, last);                    |
    |33           print_average (my_list, first, last - 3);                |
    |34       }                                                            |
    |35                                                                    |
    |36                                                                    |
    |37                                                                    |
    |38                                                                    |
    |39                                                                    |
    |40                                                                    |
    |41                                                                    |
    |42                                                                    |
    |----------------------------------------------------------------------|
  File: average.c    Procedure: ??    Line: ??      pc: ??
 Wildebeest is free software, covered by the GNU General Public License, and
 you are welcome to change it and/or distribute copies of it under certain
 conditions.  Type "show copying" to see the conditions.  There is
 absolutely no warranty for Wildebeest.  Type "show warranty" for details.
 ---Type <return> to continue, or q <return> to quit---
 Wildebeest was built for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
 ..
 (gdb)
```

The terminal window is divided into two panes: a Source pane at the top and a Command pane at the bottom. In the middle is a *locator bar* that shows the current file, procedure, line, and program counter (PC) address, when they are known to the debugger.

When you set a breakpoint on the main program by issuing the command

```
b main
```

an asterisk (*) appears opposite the first executable line of the program. When you execute the program up to the first breakpoint by issuing the command

```
run
```

a right angle bracket (>) points to the current location. So after you issue those commands, the window looks something like this:

Figure 2

```
   |----------------------------------------------------------------------|
   |27      }                                                             |
   |28                                                                    |
   |29      int main(void)                                                |
   |30      {                                                             |
   |31          /* Try two test cases. */                                 |
 *>|32          print_average (my_list, first, last);                     |
   |33          print_average (my_list, first, last - 3);                 |
   |34      }                                                             |
   |35                                                                    |
   |36                                                                    |
   |37                                                                    |
   |38                                                                    |
   |39                                                                    |
   |----------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
..
(gdb) b main
Breakpoint 1 at 0x3524: file average.c, line 32.
(gdb) run
Starting program: /home/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb)
```

15.2 Automatically running a program at startup

WDB does not start running the target executable at startup as do '`xdb`' and HP DDE. This makes it easy to set break points before the target program's main function.

To make WDB automatically start running the target program add these lines to your startup file, '`.gdbinit`':

```
break main
run
```

15.3 Screen Layouts

The TUI supports four panes within the terminal window, in various combinations:

- Command
- Source
- Disassembly
- Register

The Command pane is always present. The possible configurations of the other panes are:

- Source
- Disassembly
- Source/Disassembly
- Disassembly/Register
- Source/Register

The `layout` command (abbreviated `la`) enables you to change from one window configuration to another.

Note: You can abbreviate any command to its shortest unambiguous form.

15.3.1 Source pane

The Source pane, Figure 1, appears by default when you invoke the debugger. You can also make it appear by issuing the command

```
la src
```

15.3.2 Disassembly pane

The Disassembly pane appears when you issue the command

```
la asm
```

The pane looks like this:

Figure 3

```
   |--------------------------------------------------------------------|
   |;;;     print_average (my_list, first, last);                       |
 *>|0x3524 <main+8> addil L'-0x800,%dp,%r1                              |
   |0x3528 <main+12>        ldo 0x730(%r1),%r26                         |
   |0x352c <main+16>        ldi 9,%r24                                  |
   |0x3530 <main+20>        ldi 0,%r25                                  |
   |0x3534 <main+24>        ldil L'0x3000,%r31                          |
   |0x3538 <main+28>        be,l 0x498(%sr4,%r31)                       |
   |0x353c <main+32>        copy %r31,%rp                               |
   |;;;     print_average (my_list, first, last - 3);                   |
   |0x3540 <main+36>        addil L'-0x800,%dp,%r1                      |
   |0x3544 <main+40>        ldo 0x730(%r1),%r26                         |
   |0x3548 <main+44>        ldi 6,%r24                                  |
   |0x354c <main+48>        ldi 0,%r25                                  |
   |--------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) b main
Breakpoint 1 at 0x3524: file average.c, line 32.
(gdb) r
Starting program: /home/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb)
```

15.3.3 Source/Disassembly pane

The Source/Disassembly pane appears when you issue the command

```
la split
```

You can also reach this pane from the Source pane with the XDB command:

```
td
```

The window looks like this:

Figure 4

```
   :..................................................................:
 *>:32          print_average (my_list, first, last);                 :
   :33          print_average (my_list, first, last - 3);             :
   :34      }                                                         :
   :35                                                                :
   :36                                                                :
   :37                                                                :
   :..................................................................:
   |;;;     print_average (my_list, first, last);                     |
 *>|0x3524 <main+8> addil L'-0x800,%dp,%r1                            |
   |0x3528 <main+12>        ldo 0x730(%r1),%r26                       |
   |0x352c <main+16>        ldi 9,%r24                                |
   |0x3530 <main+20>        ldi 0,%r25                                |
   |0x3534 <main+24>        ldil L'0x3000,%r31                        |
   |------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
Breakpoint 1 at 0x3524: file average.c, line 32.
(gdb) r
Starting program: /home/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb)
```

15.3.4 Disassembly/Register pane

The Disassembly/Register pane appears when you issue the command

```
la regs
```

when the current pane is the Source/Disassembly pane. By default, the debugger displays the general registers.

The window looks like this:

Figure 5

```
:.......................................................................:
:flags 29000041         r1 51a800              rp 7f6ce597              :
:r3 7f7f0000            r4 1                   r5 7f7f06f4              :
:r6 7f7f06fc            r7 7f7f0800            r8 7f7f0800              :
:r9 40006b10            r10 0                  r11 40004b78             :
:r12 1                  r13 0                  r14 0                    :
:r15 0                  r16 40003fb8           r17 4                    :
:.......................................................................:
   |;;;     print_average (my_list, first, last);                       |
 *>|0x3524 <main+8> addil L'-0x800,%dp,%r1                              |
   |0x3528 <main+12>       ldo 0x730(%r1),%r26                          |
   |0x352c <main+16>       ldi 9,%r24                                   |
   |0x3530 <main+20>       ldi 0,%r25                                   |
   |0x3534 <main+24>       ldil L'0x3000,%r31                           |
   |--------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) r
Starting program: /home/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb)
```

15.3.5 Source/Register pane

The Source/Register pane appears when you issue the command

```
la regs
```

when the current pane is the Source pane.

The screen looks like this:

Figure 6

```
:.........................................................................:
:flags 29000041          r1 51a800               rp 7f6ce597              :
:r3 7f7f0000             r4 1                    r5 7f7f06f4              :
:r6 7f7f06fc             r7 7f7f0800             r8 7f7f0800              :
:r9 40006b10             r10 0                   r11 40004b78             :
:r12 1                   r13 0                   r14 0                    :
:r15 0                   r16 40003fb8            r17 4                    :
:.........................................................................:
 *>|32          print_average (my_list, first, last);                     |
   |33          print_average (my_list, first, last - 3);                 |
   |34      }                                                             |
   |35                                                                    |
   |36                                                                    |
   |37                                                                    |
   |----------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb)
```

15.4 Cycling through the panes

Use the commands

```
la next
```

and

```
la prev
```

to move from one pane to another without specifying a window name. If you specify `la next` repeatedly, the order the debugger uses is

- Source (`src`)
- Disassembly (`asm`)
- Source/Disassembly (`split`)
- Source/Register
- Disassembly/Register

If you invoked the `gdb` command with the `-xdb` option as well as the `-tui` option, you can also use the following commands:

`td`	Toggle between Source and Disassembly/Register panes.
`ts`	Toggle split-screen mode.

15.5 Changing pane focus

The command pane always has keyboard focus, so that you can enter debugger commands. If there is only one other pane (Source or Disassembly), the other pane has the *logical* focus, so that you can scroll within that pane by using the arrow keys or the ⟨Page Up⟩ and ⟨Page Down⟩ keys (on some keyboards these are ⟨Prev⟩ and ⟨Next⟩).

Note: In the command pane, the scrolling behavior only works for an '`hpterm`' and not for an '`xterm`' or '`dtterm`'.

If you are in split-screen mode, you may want to change the logical focus of the pane. To do so, use the command

```
focus {win_name | prev | next}
```

where *win_name* can be `src`, `asm`, `regs`, or `cmd`.

Remember, if you change the focus to a pane other than the command pane, you need to use `focus cmd` to switch back to the command pane to enter or scroll through commands.

For example, with the sequence of commands just issued, you are in split-screen mode with the focus in the Source pane.

The pane with logical focus has a border constructed from `"|"` and `"-"`.

A pane that does not have logical focus has a border constructed from `":"`vand `"."`:

Figure 7

```
:.........................................................................:
:flags 29000041          r1 51a800               rp 7f6ce597              :
:r3 7f7f0000             r4 1                    r5 7f7f06f4              :
:r6 7f7f06fc             r7 7f7f0800             r8 7f7f0800              :
:r9 40006b10             r10 0                   r11 40004b78             :
:r12 1                   r13 0                   r14 0                    :
:r15 0                   r16 40003fb8            r17 4                    :
:.........................................................................:
 *>|32          print_average (my_list, first, last);                     |
   |33          print_average (my_list, first, last - 3);                 |
   |34      }                                                             |
   |35                                                                    |
   |36                                                                    |
   |37                                                                    |
   |----------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb)
```

By default, the Source pane can scroll. To change the focus so that you can scroll in the Register pane, use the `focus` command (abbreviated `foc` or `fs`):

```
fs regs
```

or

```
foc next
```

If you then use the ⟨Page Down⟩ key to scroll in the Register pane, the window looks like this:

Figure 8

```


|--------------------------------------------------------------------------|
|flags 29000041          r1 51a800               rp 7f6ce597               |
|r3 7f7f0000             r4 1                    r5 7f7f06f4               |
|r6 7f7f06fc             r7 7f7f0800             r8 7f7f0800               |
|r9 40006b10             r10 0                   r11 40004b78              |
|r12 1                   r13 0                   r14 0                     |
|r15 0                   r16 40003fb8            r17 4                     |
|--------------------------------------------------------------------------|
 *>:32         print_average (my_list, first, last);                        :
   :33         print_average (my_list, first, last - 3);                    :
   :34      }                                                               :
   :35                                                                      :
   :36                                                                      :
   :37                                                                      :
   :......................................................................:
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb) foc next
Focus set to REGS window.
(gdb)
```

15.6 Scrolling panes

To scroll within a pane, you can use the arrow keys or the ⟨Page Up⟩ and ⟨Page Down⟩ keys (on some keyboards these are ⟨Prev⟩ and ⟨Next⟩). You can also use the following commands:

`{+ | -} [`*`num_lines`*`] [`*`win_name`*`]`
: Vertically scroll the pane forward (`+`) or backward (`-`). `+` or `-` with no arguments scrolls the pane forward or backward one page. Use *num_lines* to specify how many lines to scroll the pane. Use *win_name* to specify a pane other than the one with logical focus.

`{< | >}[`*`num_char`*`] [`*`win_name`*`]`
: Horizontally scroll the pane left (`<`) or right (`>`) the specified number of characters. If you do not specify *num_char*, the pane is scrolled one character.

Note that a space is required between the `+`, `-`, `<`, or `>` and the number.

To scroll the command pane, use the scroll bars on the terminal pane.

15.7 Changing the register display

To look at the floating-point or special registers instead of the general registers, and then to return to the general registers, you can use the following XDB commands:

`fr`
`display $fregs`
Display the floating-point registers.

`sr`
`display $sregs`
Display the special registers.

`gr`
`display $gregs`
Display the general registers.

For example, if you use the `fr` command, the window looks like this:

Figure 9

```
|--------------------------------------------------------------------------|
|flags 29000041          r1 51a800               rp 7f6ce597               |
|r3 7f7f0000             r4 1                    r5 7f7f06f4               |
|r6 7f7f06fc             r7 7f7f0800             r8 7f7f0800               |
|r9 40006b10             r10 0                   r11 40004b78              |
|r12 1                   r13 0                   r14 0                     |
|r15 0                   r16 40003fb8            r17 4                     |
   :.......................................................................:
   :30      {                                                              :
   :31          /* Try two test cases. */                                  :
 *>:32          print_average (my_list, first, last);                      :
   :33          print_average (my_list, first, last - 3);                  :
   :34      }                                                              :
   :35                                                                     :
   :.......................................................................:
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb) foc next
Focus set to REGS window.
(gdb) fr
#0  main () at average.c:32
(gdb)
```

The default floating-point register display is single-precision. To change the register display to double-precision and then back again, use the `XDB toggle float` command:

```
toggle $fregs
```

The window looks like this:

Figure 10

```
|-------------------------------------------------------------------------|
|fpsr 0                              fpe1 0                               |
|fpe2 0                              fpe3 0                               |
|fpe4 0                              fpe5 0                               |
|fpe6 0                              fpe7 0                               |
|fr4     0                           fr4R    0                            |
|fr5     1.0000000000000011          fr5R    7.00649232e-45               |
|-------------------------------------------------------------------------|
 *>:32          print_average (my_list, first, last);                     :
   :33          print_average (my_list, first, last - 3);                 :
   :34      }                                                             :
   :35                                                                    :
   :36                                                                    :
   :37                                                                    :
   :......................................................................:
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb) foc next
Focus set to REGS window.
(gdb) fr
(gdb) tf
(gdb)
```

15.8 Changing the pane size

To specify a new height for a pane or to increase or decrease the current height, use the `winheight` command (abbreviated `winh` or `wh`).

The syntax is:

```
winheight [win_name] [+ | -] num_lines
```

If you omit *win_name*, the pane with logical focus is resized. When you increase the height of a pane, the height of the Command pane is decreased by the same amount, and vice versa. The height of any other panes remains unchanged.

For example, the command

```
wh src +3
```

increases the size of the source pane, and decreases the size of the command pane, by 3 lines.

To find out the current sizes of all panes, use the `info win` command. For example, if you have a split-screen layout, the command output might be as follows:

```
(gdb) i win
        SRC     (8 lines)
        REGS    (8 lines)
        CMD     (8 lines)
```

If you use the mouse or window menus to resize the terminal window during a debugging session, the window remains the same size it was when you started. To change the window size, you must exit the debugger and restart it.

15.9 Refreshing and updating the window

If the screen display is disrupted for some reason, use the `refresh` command (`ref`) to restore the windows to their previous state:

```
ref
```

If you use stack-navigation commands such as `up`, `down`, and `frame` to change your source location, and you want to return the display to the current point of execution, use the `update` command (`upd`):

```
upd
```

16 XDB to WDB Transition Guide

This transition aid is designed for XDB users who are learning WDB, an HP-supported version of the industry-standard GDB debugger. Select one of these lists for a table that shows WDB equivalents for many common XDB commands and other features.

Invoke WDB with the command `gdb -tui` to obtain a terminal user interface (TUI) similar to that provided by XDB. Commands marked "(with `-tui`)" are valid when you use the `-tui` option.

Invoke WDB with the command `gdb -xdb` to turn on XDB compatibility mode, which enables you to use many XDB commands as synonyms for GDB commands. Commands marked "(with `-xdb`)" are valid when you use the `-xdb` option.

You may use both `-xdb` and `-tui` at the same time. Some commands are valid only when you use both options.

For a tutorial introduction to WDB, refer to the Getting Started with WDB.

- Section 16.1 [By-Function Lists of XDB Commands and HP WDB Equivalents], page 213.
- Section 16.2 [XDB Data Formats and HP WDB Equivalents], page 221.
- Section 16.3 [XDB Data Formats and HP WDB Equivalents], page 228.
- Section 16.4 [XDB Location Syntax and HP WDB Equivalents], page 230.
- Section 16.5 [XDB Special Language Operators and HP WDB Equivalents], page 231.
- Section 16.6 [XDB Special Variables and HP WDB Equivalents], page 232.
- Section 16.7 [XDB Variable Identifiers and HP WDB Equivalents], page 233.
- Section 16.8 [Alphabetical Lists of XDB Commands and HP WDB Equivalents], page 233.

16.1 By-function lists of XDB commands and HP WDB equivalents

- Section 16.1.1 [Invocation Commands], page 214.
- Section 16.1.2 [Window Mode Commands], page 214.
- Section 16.1.3 [File Viewing Commands], page 215.
- Section 16.1.4 [Source Directory Mapping Commands], page 217.
- Section 16.1.5 [Data Viewing and Modification Commands], page 217.
- Section 16.1.6 [Stack Viewing Commands], page 219.
- Section 16.1.7 [Status Viewing Command], page 219.
- Section 16.1.8 [Job Control Commands], page 220.
- Section 16.2 [Overall Breakpoint Commands], page 221.
- Section 16.2.1 [Auxiliary Breakpoint Commands], page 221.
- Section 16.2.2 [Breakpoint Creation Commands], page 222.
- Section 16.2.3 [Breakpoint Status Commands], page 223.
- Section 16.2.4 [All-Procedures Breakpoint Commands], page 224.

- Section 16.2.5 [Global Breakpoint Commands], page 225.
- Section 16.2.6 [Assertion Control Commands], page 225.
- Section 16.2.7 [Record and Playback Commands], page 225.
- Section 16.2.8 [Macro Facility Commands], page 226.
- Section 16.2.9 [Signal Control Commands], page 226.
- Section 16.2.10 [Miscellaneous Commands], page 227.

16.1.1 Invocation commands

By default, HP WDB runs in line mode. To run it with a terminal user interface similar to that of XDB, use the `-tui` option.

The following table lists the XDB and the equivalent WDB commands for invoking the terminal user interface:

XDB Command	WDB Equivalent	Meaning
`xdb` *program*	`gdb -xdb` *program*, `gdb -xdb -tui` *program*	Debug program
`xdb` *program corefile*	`gdb -xdb` *program* `-c` *corefile*	Debug core file
`xdb -d` *dir*	`gdb -xdb -d` *dir*	Specify alternate directory to search for source files
`xdb -P` *pid program*	`gdb -xdb` *program pid*	Attach to running program at invocation
`xdb -i`	(after starting) `run <` *file*	Specify input to target program
`xdb -o`	(after starting) `run >` *file*	Specify output from target program

16.1.2 Window mode commands

The following commands are TUI mode or XDB compatibility mode commands. They are available when you invoke WDB by using the `-tui` or `-xdb` or both options.

XDB Command	WDB Equivalent	Meaning

`{+ | -}r`	`{+ | -}r` (with `-xdb -tui`), `{+ | -} data` (with `-tui`)	Scroll floating-point registers forward or backward (`src`, `cmd`, and `asm` are also valid window names)
`fr`	`fr` (with `-xdb -tui`), `display $fregs` (with `-tui`)	Display floating-point registers
`gr`	`gr` (with `-xdb -tui`), `display $regs` (with `-tui`)	Display general registers
`sr`	`sr` (with `-xdb -tui`), `display $sregs` (with `-tui`)	Display special registers
`td`	`td` (with `-xdb -tui`)	Toggle disassembly mode
`tf`	`tf` (with `-xdb -tui`), `toggle $fregs` (with `-tui`)	Toggle float register display precision
`ts`	`ts` (with `-xdb -tui`)	Toggle split-screen mode
`u`	`u` (with `-xdb -tui`), `update` (with `-tui`)	Update screen to current execution point
`U`	`U` (with `-xdb -tui`), `refresh` (with `-tui`)	Refresh all windows
`w` *number*	`w` *number* (with `-xdb -tui`), `winheight src` *number* (with `-tui`)	Set size of source window

16.1.3 File viewing commands

The following table lists the XDB and the equivalent WDB commands for viewing source files:

XDB Command	WDB Equivalent	Meaning
`{+ | -}`[*number*]	`{+ | -}`[*number*] (with `-tui`; note that a space is required between `+` or `-` and the *number*)	Move view location forward or backward in source file *number* lines

`/`[*string*]	`/`[*string*] (with `-xdb`), `search` *regexp*, `forw` *regexp*	Search source forward for [last] *string*
`?`[*string*]	`?`[*string*] (with `-xdb`), `rev` *regexp*	Search source backward for [last] *string*
`D "`*dir*`"`	`D "`*dir*`"` (with `-xdb`), `dir` *pathname*	Add a directory search path for source files
`L`	`L` (with `-xdb`)	Show current viewing location or current point of execution
`ld`	`ld` (with `-xdb`), `show directories`	List source directory search path (list all directories)
`lf`	`lf` (with `-xdb`), `info sources`	List all source files
`lf` [*string*]	No equivalent	List matching files
`n`	`fo` or `rev`	Repeat previous search
`N`	`fo` or `rev`	Repeat previous search in opposite direction
`v`	`v` (with `-xdb`), `list`	Show one source window forward from current
`v` *location*	`v` *location* (with `-xdb`), `list` *location*	View source at *location* in source window
`va` *address*	`va` *address* (with `-xdb`), `disas` *address*	View *address* in disassembly window
`va` *label*	`va` *label* (with `-xdb`), `disas` *label*	View *label* in disassembly window (*label* is a location)
`va` *label* `+` *offset*	`va` *label* `+` *offset* (with `-xdb`), `disas` *label* `+` *offset*	View *label* `+` *offset* in disassembly window

16.1.4 Source directory mapping commands

Use the `D` or `dir` command to add new directories to be searched for source files. See See Section 16.1.3 [XDB-fil], page 215.

GDB does not provide a source directory mapping capability and therefore does not have any equivalent of the `apm`, `dpm`, and `lpm` commands.

16.1.5 Data Viewing and modification commands

There are many `info` commands in addition to those shown here. Use `help info` to get a list.

The following table lists the XDB and equivalent WDB commands for viewing and modifying the program data:

XDB Command	WDB Equivalent	Meaning
`l`	`l` (with `-xdb`), `info args` followed by `info locals`	List all parameters and locals of current procedure
`lc` [*string*]	`lc` [*string*] (with `-xdb`), `info common` *string*	List all (or matching) commons
`lg` [*string*]	`lg` [*string*] (with `-xdb`), `info variables` [*string*]	List all (or matching) globals
`ll` [*string*]	`info functions` [*string*], `info variables` [*string*], `maint print msymbols` *file*	List the contents of the linker symbol table
`lm`	`show user`	List all string macros
`lm` *string*	`show user` *string*	List matching string macros
`lo` [[*class*]`::`][*string*]	`info func` [[*class*]`::`][*string*]	List all (or matching) overloaded functions
`lp`	`info functions`	Show current scope, list program blocks, list names (symbols)
`lp` [[*class*]`::`]*string*	`info func` [[*class*]`::`]*string* `info addr` [[*class*]`::`]*string*	List all (or matching) procedures
`lr`	`lr` (with `-xdb`), `info all-reg`	List all registers

`lr` *string*	`lr` *string* (with `-xdb`), `info reg` *string*	List matching registers
`ls` [*string*]	No equivalent	List all (or matching) special variables
`mm`	`info sharedlibrary`	Show memory map of all loaded shared libraries
`mm` *string*	No equivalent	Show memory map of matching loaded shared libraries
`p` *expr*[`\`*format*	`p`[`/`*format* *expr* [Note: The *count* and *size* portions of formats are not allowed in the `p` (`print`) command. They are allowed in the `x` command (examine memory).]	Print value using the specified format
`p` *expr*`?`*format*	`p/`*format* `&amp;`*expr*	Print address using specified format
`p` *class*`::`	No equivalent	Print static members of *class*
`p $lang`	`show language`	Inquire what language is used
`p {+ | -}`[`\`*format*	Use `x/`*format* command to obtain initial value, then use `x` with no argument to obtain value of next memory location. To obtain value of previous memory location, use `"x $_ - 1"`.	Print value of next/previous memory location using *format*
`pq` *expr*	`set` *expr*, `set var` *expr*	Evaluate using the specified format
`pq` *expr*`?`*format*	No equivalent	Determine address using specified format
`pq` *class*`::`	No equivalent	Evaluate static members of *class*

`pq {+ | -}[\`*format*	No equivalent	Evaluate next/previous memory location using *format*

16.1.6 Stack viewing commands

The GDB concept of the top and bottom of the stack is the opposite of `XDB`, so the `XDB up` is GDB `down`.

The following table lists the XDB and equivalent WDB commands for viewing the stack contents:

XDB Command	WDB Equivalent	Meaning
`down`	`up`	View procedure one level nearer outermost frame of stack (higher number)
`down` *number*	`up` *number*	View procedure *number* levels nearer outermost frame of stack
`t` [*depth*]	`t` [*depth*] (with `-xdb`), `bt` [*depth*]	Print stack trace to *depth*
`T` [*depth*]	`T` [*depth*] (with `-xdb`), `bt full` [*depth*]	Print stack trace and show local vars
`top`	`frame 0`	View procedure at innermost frame of stack
`up`	`down`	View procedure one level nearer innermost frame of stack (lower number)
`up` *number*	`down` *number*	View procedure *number* levels nearer innermost frame of stack
`V` [*depth*]	`V` [*depth*] (with `-xdb`), `frame` [*depth*]	Display text for current active procedure or at specified *depth* on stack

16.1.7 Status-viewing command

Type the `show` command with no arguments to get a list of current debugger settings.

XDB Command	WDB Equivalent	Meaning
`I`	`info` (many kinds), `show` (many kinds)	Display state of debugger and program

16.1.8 Job control commands

The following table lists the XDB and equivalent WDB commands for controlling program execution:

XDB Command	WDB Equivalent	Meaning
`c`	`c`, `continue`	Continue from breakpoint, ignoring any pending signal
`c` *location*	`until` *location*	Continue from breakpoint, ignoring any pending signal, set temporary breakpoint at *location*
`C`	`c`, `continue`	Continue, allowing any pending signal
`C` [*location*]	`until` *location*	Continue, allowing any pending signal, set temporary breakpoint at *location*
`g` *line*	`g` *line* (with `-xdb`), `go` *line*, `tb` *line* followed by `jump` *line*	Go to *line* in current procedure
`g #`*label*	No equivalent	Go to *label* in current procedure
`g {+ | -}`*lines*	`g {+ | -}`*lines* (with `-xdb`), `go {+ | -}`*lines*, `tb {+ | -}`*lines* followed by `jump {+ | -}`*lines*	Go forward or back given # lines
`g {+ | -}`	`g {+ | -}` (with `-xdb`), `go {+ | -}1`, `tb {+ | -}1` followed by `jump {+ | -}1`	Go forward or back 1 line
`k`	`k`	Detach and terminate target

`r` [*arguments*]	`r` [*arguments*]	Run with last arguments [or with new arguments]
`R`	`R` (with `-xdb`), `r`	Rerun with no arguments
`s`	`s`, `si`	Single step (into procedures) (`si`: step by instruction)
`s` *number*	`s` *number*, `si` *number*	Single step *number* steps (into procedures) (`si`: step by instruction)
`S`	`S` (with `-xdb`), `n`, `ni`	Step over (`ni`: step over by instruction)
`S` *number*	`S` *number* (with `-xdb`), `n` *number*, `ni`*number*	Step over by *number* statements or instructions (`ni`: step over by instruction)

16.2 Overall breakpoint commands

The following table lists the XDB and equivalent WDB commands for setting additional breakpoints:

XDB Command	WDB Equivalent	Meaning
`lb`	`lb` (with `-xdb`), `i b`	List breakpoints
`tb`	No equivalent	Toggle overall breakpoint state

16.2.1 Auxiliary breakpoint commands

The following table lists the XDB and equivalent WDB auxiliary breakpoint related commands:

XDB Command	WDB Equivalent	Meaning

`"`*any_string*`"`	p `"`*any_string*`"`	Print *any_string*
`if` *expr* `{`*cmds*`}` [`{`*cmds*`}`]	`if` *expr cmds* [`else` *cmds*] `end`	Conditionally execute *cmds*
`Q`	`Q` (with `-xdb`), `silent` (must be first command in a commands list)	Quiet breakpoints

16.2.2 Breakpoint creation commands

The GDB equivalent of the `count` and `cmds` arguments is to use the `commands`*bnum* command to set an ignore count and/or to specify commands to be executed for that breakpoint.

For C++ programs, you can use the regular-expression breakpoint command `rbreak` to set breakpoints on all the member functions of a class or on overloaded functions outside a class.

The following table lists the XDB and equivalent WDB commands for creating breakpoints:

XDB Command	WDB Equivalent	Meaning
`b` *loc*	`b` *loc*	Set a breakpoint at the specified location
`b`	`b`	Set a breakpoint at the current line
`ba` *address*	`ba` *address* (with `-xdb`), `b *`*address*	Set breakpoint at a code address
`bb` [*depth*]	No equivalent (use `b` *proc*)	Set breakpoint at procedure beginning
`bi` *expr*`.`*proc*	`b` *class*`::`*proc* `cond` *bnum* `(this == `*expr*`)`	Set an instance breakpoint at the first executable line of *expr*`.`*proc*
`bi -c` *expr*	No equivalent	Set an instance breakpoint at first executable line of all non-static member functions of the instance of a class (no base classes)

`bi -C` *expr*	No equivalent	Set an instance breakpoint at first executable line of all non-static member functions of the instance's class (base classes included)
`bpc -c` *class*	`rb ^`*class*`::*`	Set a class breakpoint at first executable line of all member functions of the instance's class (no base classes)
`bpc -C` *class*	Use `rb ^`*class*`::*` for base classes also	Set a class breakpoint at first executable line of all member functions of the class (base classes included)
`bpo` *proc*	`rb` *proc*	Set breakpoints on overloaded functions outside a class
`bpo` *class*`::`*proc*	`b` *class*`::`*proc*	Set breakpoints on overloaded functions in a class
`bt` [*depth*]	No equivalent	Set trace breakpoint at procedure at specified *depth* on program stack
`bt` *proc*	`b` *proc* `commands` *bnum* `finish c end`	Set trace breakpoint at *proc*
`bu` [*depth*]	`bu` [*depth*] (with `-xdb`). The `finish` command is equivalent to the sequence `bu`, `c`, `db` (to continue out of the current routine).	Set up-level breakpoint
`bx` [*depth*]	`bx` [*depth*] (with `-xdb`)	Set a breakpoint at procedure exit

16.2.3 Breakpoint status commands

The following table lists the XDB and equivalent WDB commands for changing the breakpoint status:

XDB Command	WDB Equivalent	Meaning
`ab` *number*	`enable` *number*	Activate suspended breakpoint of the given *number*
`ab *`	`enable`	Activate all suspended breakpoints
`ab @`*shared-library*	No equivalent	Activate suspended breakpoints in named shared library
`bc` *number expr*	`bc` *number expr* (with `-xdb`), `ignore`*number expr* (within a commands list)	Set a breakpoint count
`db`	`clear`	Delete breakpoint at current line
`db` *number*	`delete` *number*	Delete breakpoint of the given *number*
`db *`	`delete`	Delete all breakpoints
`sb` *number*	`disable` *number*	Suspend breakpoint of the given *number*
`sb *`	`disable`	Suspend all breakpoints
`sb @`*shared-library*	No equivalent	Suspend breakpoints in named shared library

16.2.4 All-procedures breakpoint commands

GDB does not provide the ability to set breakpoints on all procedures with a single command. Therefore, it does not have any equivalent of the following commands:

```
bp
bpt
bpx
dp
Dpt
```

```
Dpx
```

16.2.5 Global breakpoint commands

The following table lists the XDB and equivalent WDB commands for setting global breakpoints:

XDB Command	WDB Equivalent	Meaning
`abc` *cmds*	No exact equivalent, but `display` *expr* is equivalent to `abc print` *expr*	Set or delete *cmds* to execute at every stop
`dbc`	`undisplay`	Stop displaying values at each stop

16.2.6 Assertion control commands

GDB does not provide the ability to trace by instruction. Watchpoints, however, provide similar functionality to `xdb` assertions.

For example, watchpoints can be:

- Enabled (corresponds to `aa`)
- Disabled (corresponds to `da`)
- Listed (corresponds to `info watch`)
- Added (corresponds to `x`)

WDB does not have explicit equivalents for the following commands:

```
a
aa
da
la
sa
ta
x
```

16.2.7 Record and playback commands

Use the `source` command to read commands from a file. GDB does not provide a recording capability like XDB's, but you can use the `set history save` command to record all GDB commands in the file `./.gdb_history` (similar to the `$HOME/.xdbhist` file). The history file is not saved until the end of your debugging session.

To change the name of the history file, use `set history filename`.

To stop recording, use `set history save off`.

To display the current history status, use `show history`.

For an equivalent of the XDB record-all facility, pipe the output of the `gdb` command to the *tee*(1) command. For example:

```
gdb a.out | tee mylogfile
```

This solution works with the default line-mode user interface, not with the terminal user interface.

16.2.8 Macro facility commands

Use the `show user` or `help user-defined` command to obtain a list of all user-defined commands.

The following table lists the XDB and the equivalent WDB commands for handling macros:

XDB Command	WDB Equivalent	Meaning
`def` *name replacement-text*	`def` *name* [GDB prompts for commands]	Define a user-defined command
`tm`	No equivalent	Toggle the macro substitution mechanism
`undef` *name*	`def` *name* [follow with empty command list]	Remove the macro definition for *name*
`undef *`	No equivalent	Remove all macro definitions

16.2.9 Signal control commands

The following table lists the XDB and equivalent WDB commands for signal control:

XDB Command	WDB Equivalent	Meaning
`lz`	`lz` (with `-xdb`), `info signals`	List signal handling
`z` *number* `s`	`z` *number* `s` (with `-xdb`), `handle` *number*`stop`, `handle` *number* `nostop`	Toggle stop flag for signal *number*
`z` *number* `i`	`z` *number* `i` (with `-xdb`), `handle` *number*`nopass`, `handle` *number* `pass`	Toggle ignore flag for signal *number*

`z` *number* `r`	`z` *number* `r` (with `-xdb`), `handle` *number* `print`, `handle` *number* `noprint`	Toggle report flag for signal *number*
`z` *number* `Q`	`z` *number* `Q` (with `-xdb`), `handle` *number* `noprint`	Do not print the new state of the signal

16.2.10 Miscellaneous commands

Some of the additional XDB and the equivalent WDB commands are discussed below:

XDB Command	WDB Equivalent	Meaning
`Return`	*`Return`*	Repeat previous command
`~`	*`Return`*	Repeat previous command
`;`	No equivalent (one command per line in command list)	Separate commands in command list
`!` *cmd_line*	`!` *cmd_line* (with `-xdb`), `she` *cmd_line*	Invoke a shell
`{`*cmd_list*`}`	`commands` [*number*] `... end`	Execute command list (group commands)
Control-C	Control-C	Interrupt the program
`#` [*text*]	`#` [*text*]	A comment
`am`	`am` (with `-xdb`), `set height` *num*	Activate more (turn on pagination)
`f` [`"`*printf-style-fmt*`"`]	No equivalent	Set address printing format
`h`	`h`	Help
`M`[`{t	c}` [*expr*[`;` *expr*...]]]	No equivalent	Print object or corefile map

`q`	`q`	Quit debugger
`sm`	`sm` (with `-xdb`), `set height 0`	Suspend more (turn off pagination)
`ss` *file*	No equivalent	Save (breakpoint, macro, assertion) state
`tc`	No equivalent	Toggle case sensitivity in searches

16.3 XDB data formats and HP WDB equivalents

The format of the `print` command is different in XDB and GDB:

```
XDB:  p expr\fmt
GDB:  p/fmt expr
```

Use the `set print pretty` command to obtain a structured display similar to the default XDB display.

The following table lists the XDB and equivalent WDB commands for setting data display formats:

XDB Command	WDB Equivalent	Meaning
`b`	`d`	Byte in decimal
`B` (1)	`d`	Byte in decimal
`c`	`c`	Character
`C` (1)	`c`	Wide character
`d`	`d`	Decimal integer
`D` (1)	`d`	Long decimal integer
`e`	No equivalent	`e` floating-point notation as float

`E` (1)	No equivalent	`e` floating-point notation as double
`f`	No equivalent	`f` floating-point notation as float
`F` (1)	No equivalent	`f` floating-point notation as double
`g`	`f`	`g` floating-point notation as float
`G` (1)	`f`	`g` floating-point notation as double
`i`	Use `x/i` command	Machine instruction (disassembly)
`k`	No equivalent	Formatted structure display
`K` (1)	No equivalent	Formatted structure display with base classes
`n`	`print`	Normal (default) format, based on type
`o`	`o`	Expression in octal as integer
`O` (1)	`o`	Expression in octal as long integer
`p`	`a`	Print name of procedure containing address
`s`	No equivalent	String
`S`	No equivalent	Formatted structure display

`t`	`whatis`, `ptype`	Show type of the expression
`T` (1)	`ptype`	Show type of expression, including base class information
`u`	`u`	Expression in unsigned decimal format
`U` (1)	`u`	Expression in long unsigned decimal format
`w`	No equivalent	Wide character string
`W` (1)	No equivalent	Address of wide character string
`x`	`x`	Print in hexadecimal
`X` (1)	`x`	Print in long hexadecimal
`z`	`t`	Print in binary
`Z` (1)	`t`	Print in long binary

(1) HP WDB will display data in the size appropriate for the data. It will not extend the length displayed in response to one of the uppercase formchars (e.g. `O`, `D`, `F`).

16.4 XDB location syntax and HP WDB equivalents

The following command lists the XDB and the equivalent WDB commands for locating source lines:

XDB Location Syntax	WDB Equivalent	Meaning
line	*line*	Source line and code address
file[`:`*line*]	*file*[`:`*line*]	Source line and code address

proc	*proc*	Procedure name
[*file*:]*proc*[:*proc*[...]][:*line*]	No equivalent	Source line and code address
[*file*:]*proc*[:*proc*[...]][:#*label*]	No equivalent	Source line and code address
[*class*]::*proc*	[*class*]::*proc*	Source line and code address
[*class*]::*proc*[:*line*]	No equivalent	Source line and code address
[*class*]::*proc*[#*label*]	No equivalent	Source line and code address
proc#*line*	No equivalent	Code address
[*class*]::*proc*#*line*	No equivalent	Code address
#*label*	No equivalent	Source line and code address
name@*shared-library*	No equivalent	Address of *name* in shared library *shared-library*

16.5 XDB special language operators and HP WDB equivalents

The following table lists the XDB and the equivalent WDB commands for language operators:

XDB Language Operator	WDB Equivalent	Meaning
`$addr`	Depends on language	Unary operator, address of object
`$in`	No equivalent	Unary Boolean operator, execution in procedure
`$sizeof`	`sizeof`	Unary operator, size of object

16.6 XDB special variables and HP WDB equivalents

GDB does not provide special variables of the kind that XDB has, but you can use `show` and `set` to display and modify many debugger settings.

XDB Special Variable	WDB Equivalent	Meaning
`$cplusplus`	No equivalent	C++ feature control flags
`$depth`	No equivalent	Default stack depth for local variables
`$fpa`	No equivalent	Treat FPA sequence as one instruction
`$fpa_reg`	No equivalent	Address register for FPA sequences
`$lang`	`show language`	Current language for expression evaluation
`$line`	No equivalent	Current source line number
`$malloc`	No equivalent	Debugger memory allocation (bytes)
`$print`	No equivalent	Display mode for character data
`$`*regname*	`$`*regname*	Hardware registers
`$result`	Use `$`*n* (value history number assigned to the desired result)	Return value of last command line procedure call
`$signal`	No equivalent	Current child procedure signal number

`$step`	No equivalent	Number of instructions debugger will step in non-debuggable procedures before free-running
`$`*var*	`$`*var*	Define or use special variable (convenience variable)

16.7 XDB variable identifiers and HP WDB equivalents

XDB Variable Identifier	WDB Equivalent	Meaning
var	*var*	Search for *var*
class`::`*var*	*class*`::`*var*	Search *class* for *var* (bug: not yet)
[[*class*]`::`]*proc*`:`[*class*`::`]*var*	*proc*`::`*var*	Search *proc* for *var* (static variables only)
[[*class*]`::`]*proc*`:`*depth*`:`[*class*`::`	No equivalent	Search *proc* for depth on stack
`. (dot)`	Empty string; for example, `p` is the equivalent of `p .`	Shorthand for last thing you looked at
`:`*var* or `::`*var*	`::`*var* to distinguish a global from a local variable with same name	Search for global variable only

16.8 Alphabetical lists of XDB commands and HP WDB equivalents

- Section 16.8.1 [A], page 234.
- Section 16.8.2 [B], page 234.
- Section 16.8.3 [C though D], page 236.
- Section 16.8.4 [F through K], page 237.
- Section 16.8.5 [L], page 238.
- Section 16.8.6 [M through P], page 239.
- Section 16.8.7 [Q through S], page 240.
- Section 16.8.8 [T], page 242.

- Section 16.8.9 [U through Z], page 242.
- Section 16.8.10 [Symbols], page 244.

16.8.1 A

XDB Command	Equivalent WDB Command
`a` [*cmds*]	No equivalent
`aa` *number*	No equivalent
`aa *`	No equivalent
`ab` *number*	`enable` *number*
`ab *`	`enable`
`ab @`*shared-library*	No equivalent
`abc` *cmds*	No exact equivalent, but `display` *expr* is equivalent to `abc print` *expr*
`am`	`am` (with `-xdb`), `set height` *num*
`apm` *oldpath* [*newpath*]	No equivalent
`apm ""` [*newpath*]	No equivalent

16.8.2 B

XDB Command	Equivalent WDB Command
`b` *loc*	`b` *loc*

`b`	`b`
`ba` *address*	`ba` *address* (with `-xdb`), `b *`*address*
`bb` [*depth*]	No equivalent (use `b` *proc*)
`bc` *number expr*	`bc` *number expr* (with `-xdb`), `ignore` *number expr* (within a commands list)
`bi` *expr*`.`*proc*	`b` *class*`::`*proc* `cond` *bnum* `(this ==` *expr*`)`
`bi -c` *expr*	No equivalent
`bi -C` *expr*	No equivalent
`bp`	No equivalent
`bp` *cmds*	No equivalent
`bpc -c` *class*	`rb ^`*class*`::*`
`bpc -C` *class*	Use `rb ^`*class*`::*` for base classes also
`bpo` *proc*	`rb` *proc*
`bpo` *class*`::`*proc*	`b` *class*`::`*proc*
`bpt`	No equivalent
`bpt` *cmds*	No equivalent
`bpx`	No equivalent

`bpx` *cmds*	No equivalent
`bt` [*depth*]	No equivalent

`bt` *proc*	`b` *proc* `commands` *bnum*
	`finish`
	`c`
	`end`

`bu` [*depth*]	`bu` [*depth*] (with `-xdb`). The `finish` command is equivalent to the sequence `bu`, `c`, `db` (to continue out of the current routine).
`bx` [*depth*]	`bx` [*depth*] (with `-xdb`)

16.8.3 C through D

XDB Command	Equivalent WDB Command
`c`	`c`, `continue`
`c` *location*	`until` *location*
`C`	`c`, `continue`
`C` *location*	`until` *location*
`D "`*dir*`"`	`D "`*dir*`"` (with `-xdb`), `dir` *pathname*
`da` *number*	No equivalent
`da *`	No equivalent
`db`	`clear`

`db` *number*	`delete` *number*
`db *`	`delete`
`dbc`	`undisplay`
`def` *name replacement-text*	`def` *name* [GDB prompts for commands]
`down`	`up`
`down` *number*	`up` *number*
`dp`	No equivalent
`dpm` *index*	No equivalent
`dpm *`	No equivalent
`Dpt`	No equivalent
`Dpx`	No equivalent

16.8.4 F through K

XDB Command	Equivalent WDB Command
`f` ["*printf-style-fmt*"]	No equivalent
`fr`	`fr` (with `-xdb -tui`), `display $fregs` (with `-tui`)
`g` *line*	`g` *line* (with `-xdb`), `go` *line*, `tb` *line* followed by `jump` *line*

`g #`*label*	No equivalent
`g {+ | -}`*lines*	`g {+ | -}`*lines* (with `-xdb`), `go {+ | -}`*lines* `tb {+ | -}`*lines* followed by `jump {+ | -}`*lines*
`g {+ | -}`	`g {+ | -}` (with `-xdb`), `go {+ | -}1`, `tb {+ | -}1` followed by `jump {+ | -}1`
`gr`	`gr` (with `-xdb -tui`), `display $regs` (with `-tui`)
`h`	`h`
`if` *expr* `{`*cmds*`}` [`{`*cmds*`}`]	`if` *expr* *cmds* [`else` *cmds*] `end`
`I`	`info` (many kinds), `show` (many kinds)
`k`	`k`

16.8.5 L

XDB Command	Equivalent WDB Command
`l`	`l` (with `-xdb`), `info args` followed by `info locals`
`L`	`L` (with `-xdb`)
`la`	No equivalent
`lb`	`lb` (with `-xdb`), `i b`
`lc` [*string*]	`lc` [*string*] (with `-xdb`), `info common` *string*

`ld`	`ld` (with `-xdb`), `show directories`
`lf`	`lf` (with `-xdb`), `info sources`
`lf` [*string*]	No equivalent
`lg` [*string*]	`lg` [*string*] (with `-xdb`), `info variables` [*string*]
`ll` [*string*]	`info functions` [*string*], `info variables` [*string*], `maint print msymbols` *file*
`lm` [*string*]	`show user` [*string*]
`lo` [[*class*]`::`][*string*]	`info func` [[*class*]`::`][*string*]
`lp`	`info functions`
`lp` [[*class*]`::`]*string*	`info func` [[*class*]`::`]*string* `info addr` [[*class*]`::`]*string*
`lpm`	No equivalent
`lr`	`lr` (with `-xdb`), `info all-reg`
`lr` *string*	`lr` *string* (with `-xdb`), `info reg` *string*
`ls` [*string*]	No equivalent
`lz`	`lz` (with `-xdb`), `info signals`

16.8.6 M through P

XDB Command	Equivalent WDB Command

`M[{t | c} [`*expr*`[; `*expr*`...]]]`	No equivalent
`mm`	`info sharedlibrary`
`mm` *string*	No equivalent
`N`	`fo` or `rev`
`n`	`fo` or `rev`
`p` *expr*[\format]	`p`[/*format*] *expr* [Note: The *count* and *size* portions of formats are not allowed in the `p` (`print`) command. They are allowed in the `x` command (examine memory).]
`p` *expr*?*format*	`p/`*format* `&amp;`*expr*
`p` *class*`::`	No equivalent
`p $lang`	`show language`
`p {+ | -}`[\format	Use `x/`*format* command to obtain initial value, then use `x` with no argument to obtain value of next memory location. To obtain value of previous memory location, use `"x $_ - 1"`.
`pq` *expr*	`set` *expr*, `set var` *expr*
`pq` *expr*?*format*	No equivalent
`pq` *class*`::`	No equivalent
`pq` [`+ | -`][\format	No equivalent

16.8.7 Q through S

XDB Command	Equivalent WDB Command
`q`	`q`
`Q`	`Q` (with `-xdb`), `silent` (must be first command in a commands list)
`r` [*arguments*]	`r` [*arguments*]
`R`	`R` (with `-xdb`), `r`
`s`	`s`, `si`
`s` *number*	`s` *number*, `si` *number*
`S`	`S` (with `-xdb`), `n`, `ni`
`S` *number*	`S` *number* (with `-xdb`), `n` *number*, `ni`*number*
`sa` *number*	No equivalent
`sa *`	No equivalent
`sb` *number*	`disable` *number*
`sb *`	`disable`
`sb @`*shared-library*	No equivalent
`sm`	`sm` (with `-xdb`), `set height 0`
`sr`	`sr` (with `-xdb -tui`), `display $sregs` (with `-tui`)

`ss` *file*	No equivalent

16.8.8 T

XDB Command	Equivalent WDB Command
`t` [*depth*]	`t` [*depth*] (with `-xdb`), `bt` [*depth*]
`T` [*depth*]	`T` [*depth*] (with `-xdb`), `bt full` [*depth*]
`ta`	No equivalent
`tb`	No equivalent
`tc`	No equivalent
`td`	`td` (with `-xdb -tui`)
`tf`	`tf` (with `-xdb -tui`), `toggle $fregs` (with `-tui`)
`tm`	No equivalent
`top`	`frame 0`
`tr` [`@`]	No equivalent
`ts`	`ts` (with `-xdb -tui`)

16.8.9 U through Z

XDB Command	Equivalent WDB Command

`u`	`u` (with `-xdb -tui`), `update` (with `-tui`)
`U`	`U` (with `-xdb -tui`), `refresh` (with `-tui`)
`undef` *name*	`def` *name* [follow with empty command list]
`undef *`	No equivalent
`up`	`down`
`up` *number*	`down` *number*
`v`	`v` (with `-xdb`), `list`
`v` *location*	`v` *location* (with `-xdb`), `list` *location*
`V` [*depth*]	`V` [*depth*] (with `-xdb`), `frame` [*depth*]
`va` *address*	`va` *address* (with `-xdb`), `disas` *address*
`va` *label*	`va` *label* (with `-xdb`), `disas` *label*
`va` *label* `+` *offset*	`va` *label* `+` *offset* (with `-xdb`), `disas` *label* `+` *offset*
`w` *number*	`w` *number* (with `-xdb -tui`), `winheight src` *number* (with `-tui`)
`x` [*expr*]	No equivalent
`xdb` *program*	`gdb -xdb` *program*, `gdb -xdb -tui` *program*
`xdb` *program corefile*	`gdb -xdb` *program* `-c` *corefile*

`xdb -d` *dir*	`gdb -xdb -d` *dir*
`xdb -P` *pid program*	`gdb -xdb` *program pid*
`xdb -i`	(after starting) `run <` *file*
`xdb -o`	(after starting) `run >` *file*
`z` *number* `s`	`z` *number* `s` (with `-xdb`), `handle` *number* `stop`, `handle` *number* `nostop`
`z` *number* `i`	`z` *number* `i` (with `-xdb`), `handle` *number* `nopass`, `handle` *number* `pass`
`z` *number* `r`	`z` *number* `r` (with `-xdb`), `handle` *number* `print`, `handle` *number* `noprint`
`z` *number* `Q`	`z` *number* `Q` (with `-xdb`), `handle` *number* `noprint`

16.8.10 Symbols

XDB Symbol	Equivalent HP WDB Symbol
line	*line*
file[`:`*line*]	*file*[`:`*line*]
proc	*proc*
[*file*`:`]*proc*[`:`*proc*[...]][`:`*line*]	No equivalent
[*file*`:`]*proc*[`:`*proc*[...]][`:#`*label*]	No equivalent

[*class*]`::`*proc*	[*class*]`::`*proc*
[*class*]`::`*proc*[`:`*line*]	No equivalent
[*class*]`::`*proc*[`#`*label*]	No equivalent
proc`#`*line*	No equivalent
[*class*]`::`*proc*`#`*line*	No equivalent
name`@`*shared-library*	No equivalent
var	*var*
class`::`*var*	*class*`::`*var* (bug: not yet)
[[*class*]`::`]*proc*`:`[*class*`::`]*var*	*proc*`::`*var* (static variables only)
[[*class*]`::`]*proc*`:`*depth*`:`[*class*`::`]*var*	No equivalent
Return	*Return*
`"`*any_string*`"`	p `"`*any_string*`"`
`.` (dot)	Empty string; for example, `p` is the equivalent of `p .`
`~`	*Return*
`{+ | -}r`	`{+ | -}r` (with `-xdb -tui`), `{+ | -} data` (with `-tui`)

`{+ | -}`[*number*]	`{+ | -}`[*number*] (with `-tui`; note that a space is required between `+` or `-` and the *number*)
`/`[*string*]	`/`[*string*] (with `-xdb`), `search` *regexp*, `forw` *regexp*
`?`[*string*]	`?`[*string*] (with `-xdb`), `rev` *regexp*
`;`	No equivalent (one command per line in command list)
`:`*var* or `::`*var*	`::`*var*
`!` *cmd_line*	`!` *cmd_line* (with `-xdb`), `she` *cmd_line*
`{`*cmd_list*`}`	`commands` [*number*] `...` `end`
`<`*file*	`source` *file*
`<<`*file*	No equivalent
`>`	No equivalent
`>`*file*	No equivalent
`>c`	No equivalent
`>f`	No equivalent
`>t`	No equivalent
`>@`[`c | f | t`]	No equivalent

`>@`*file*	No equivalent
`>>`	No equivalent
`>>`*file*	No equivalent
`>>@`	No equivalent
`>>@`*file*	No equivalent
Control-C	Control-C
`#` [*text*]	`#` [*text*]
`#`*label*	No equivalent

17 Controlling GDB

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, see Section 8.7 [Print settings], page 70. Other settings are described here.

17.1 Setting the GDB Prompt

GDB indicates its readiness to read a command by printing a string called the *prompt*. This string is normally '`(gdb)`'. You can change the prompt string with the `set prompt` command. For instance, when debugging GDB with GDB, it is useful to change the prompt in one of the GDB sessions so that you can always tell which one you are talking to.

Note: `set prompt` does not add a space for you after the prompt you set. This allows you to set a prompt which ends in a space or a prompt that does not.

`set prompt` *`newprompt`*
: Directs GDB to use *newprompt* as its prompt string henceforth.

`show prompt`
: Prints a line of the form: '`Gdb's prompt is:` *`your-prompt`*'

17.2 Setting Command Editing Options in GDB

GDB reads its input commands via the *readline* interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or *vi*-style inline editing of commands, `csh`-like history substitution, and a storage and recall of command history across debugging sessions.

You may control the behavior of command line editing in GDB with the command `set`.

`set editing`
`set editing on`
: Enable command line editing (enabled by default).

`set editing off`
: Disable command line editing.

`show editing`
: Show whether command line editing is enabled.

17.3 Setting Command History Feature in GDB

GDB can keep track of the commands you type during your debugging sessions, so that you can be certain of precisely what happened. Use these commands to manage the GDB command history facility.

To make command history understand your `vi` key bindings you need to create a '`~/.inputrc`' file with the following contents:

```
set editing-mode vi
```

The *readline* interface uses the '`.inputrc`' file to control the settings.

`set history filename` *fname*
: Set the name of the GDB command history file to *fname*. This is the file where GDB reads an initial command history list, and where it writes the command history from this session when it exits. You can access this list through history expansion or through the history command editing characters listed below. This file defaults to the value of the environment variable `GDBHISTFILE`, or to '`./.gdb_history`' ('`./_gdb_history`' on MS-DOS) if this variable is not set.

`set history save`
`set history save on`
: Record command history in a file, whose name may be specified with the `set history filename` command. By default, this option is disabled.

`set history save off`
: Stop recording command history in a file.

`set history size` *size*
: Set the number of commands which GDB keeps in its history list. This defaults to the value of the environment variable `HISTSIZE`, or to 256 if this variable is not set.

History expansion assigns special meaning to the character `!`.

Since `!` is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the `set history expansion on` command, you may sometimes need to follow `!` (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities do not attempt substitution on the strings `!=` and `!(`, even when history expansion is enabled.

The commands to control history expansion are:

`set history expansion on`
`set history expansion`
: Enable history expansion. History expansion is off by default.

`set history expansion off`
: Disable history expansion.

: The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with GNU Emacs or `vi` may wish to read it.

`show history`
`show history filename`
`show history save`
`show history size`
`show history expansion`
: These commands display the state of the GDB history parameters. `show history` by itself displays all four states.

`show commands`
: Display the last ten commands in the command history.

`show commands n`
: Print ten commands centered on command number *n*.

`show commands +`
: Print ten commands just after the commands last printed.

17.4 Setting the GDB Screen Size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type ⟨RET⟩ when you want to continue the output, or `q` to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally GDB knows the size of the screen from the terminal driver software. For example, on Unix, GDB uses the termcap data base together with the value of the `TERM` environment variable and the `stty rows` and `stty cols` settings. If this is not correct, you can override it with the `set height` and `set width` commands:

`set height lpp`
`show height`
`set width cpl`
`show width`
: These `set` commands specify a screen height of *lpp* lines and a screen width of *cpl* characters. The associated `show` commands display the current settings.

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

  Likewise, you can specify '`set width 0`' to prevent GDB from wrapping its output.

17.5 Supported Number Formats

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with '`0`', decimal numbers end with '`.`', and hexadecimal numbers begin with '`0x`'. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers—when no particular format is specified—is base 10. You can change the default base for both input and output with the `set radix` command.

`set input-radix base`
: Set the default base for numeric input. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix; for example, any of

```
set radix 012
set radix 10.
set radix 0xa
```

sets the base to decimal. On the other hand, '`set radix 10`' leaves the radix unchanged no matter what it was.

`set output-radix` *`base`*
: Set the default base for numeric display. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix.

`show input-radix`
: Display the current default base for numeric input.

`show output-radix`
: Display the current default base for numeric display.

17.6 Optional warnings and messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the `set verbose` command. This makes GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by `set verbose` are those which announce that the symbol table for a source file is being read; see `symbol-file` in Section 12.1 [Commands to specify files], page 103.

`set verbose on`
: Enables GDB output of certain informational messages.

`set verbose off`
: Disables GDB output of certain informational messages.

`show verbose`
: Displays whether `set verbose` is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful (see Section 12.3 [Errors reading symbol files], page 107).

`set complaints` *`limit`*
: Permits GDB to output *limit* complaints about each type of unusual symbols before becoming silent about the problem. Set *limit* to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

`show complaints`
: Displays how many symbol complaints GDB is permitted to produce.

By default, GDB is cautious, and asks the user to confirm on certain commands. For example, if you try to run a program which is already running:

```
((gdb)) run
The program being debugged has been started already.
```

```
     Start it from the beginning? (y or n)
```

If you are willing to unflinchingly face the consequences of your own commands, you can disable this "feature":

`set confirm off`
: Disables confirmation requests.

`set confirm on`
: Enables confirmation requests (the default).

`show confirm`
: Displays state of confirmation requests.

17.7 Optional messages about internal happenings

`set debug arch`
: Turns on or off display of `gdbarch` debugging info. The default is off

`show debug arch`
: Displays the current state of displaying `gdbarch` debugging information.

`set debug event`
: Turns on or off display of GDB event debugging information. The default is off.

`show debug event`
: Displays the current state of displaying GDB event debugging info.

`set debug expression`
: Turns on or off display of GDB expression debugging information. The default is off.

`show debug expression`
: Displays the current state of displaying GDB expression debugging info.

`set debug overload`
: Turns on or off display of GDB C++ overload debugging info. This includes info such as ranking of functions, etc. The default is off.

`show debug overload`
: Displays the current state of displaying GDB C++ overload debugging info.

`set debug remote`
: Turns on or off display of reports on all packets sent back and forth across the serial line to the remote machine. The info is printed on the GDB standard output stream. The default is off.

`show debug remote`
: Displays the state of display of remote packets.

`set debug serial`
: Turns on or off display of GDB serial debugging info. The default is off.

`show debug serial`
: Displays the current state of displaying GDB serial debugging info.

`set debug target`
: Turns on or off display of GDB target debugging info. This info includes what is going on at the target level of GDB, as it happens. The default is off.

`show debug target`
: Displays the current state of displaying GDB target debugging info.

`set debug varobj`
: Turns on or off display of GDB variable object debugging info. The default is off.

`show debug varobj`
: Displays the current state of displaying GDB variable object debugging info.

18 Canned Sequences of Commands

In addition to breakpoint commands (see Section 5.1.6 [Breakpoint command lists], page 41), GDB provides the following two ways to store sequence of commands for execution as a unit:

- user-defined commands
- command files

18.1 User-defined commands

A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command. This is done with the `define` command. User commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command via *$arg0...$arg9*. The following example illustrates the use of canned sequence of commands:

```
define adder
  print $arg0 + $arg1 + $arg2
```

To execute the command use:

```
adder 1 2 3
```

This defines the command `adder`, which prints the sum of its three arguments. Note the arguments are text substitutions, so they may reference variables, use complex expressions, or even perform further functions calls.

The following contructs can be used to create canned sequence of commands:

`define` *commandname*
: Define a command named *commandname*. If there is already a command by that name, you are asked to confirm that you want to redefine it.

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

`if`
: Takes a single argument, which is an expression to evaluate. It is followed by a series of commands that are executed only if the expression is true (nonzero). The `if` clause can be followed by an optional `else` clause. You can add a list of commands to the `else` clause which get executed only if the expression is false.

`while`
: The syntax is similar to `if`: the command takes a single argument, which is an expression to evaluate, and must be followed by the commands to execute, one per line, terminated by an `end`. The commands are executed repeatedly as long as the expression evaluates to true.

`document` *commandname*
: Document the user-defined command *commandname*, so that it can be accessed by `help`. The command *commandname* must already be defined. This command reads lines of documentation just as `define` reads the lines of the command definition, ending with `end`. After the `document` command is finished, `help` on command *commandname* displays the documentation you have written.

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

`help user-defined`
: List all user-defined commands, with the first line of the documentation (if any) for each.

`show user`
`show user` *`commandname`*
: Display the GDB commands used to define *commandname* (but not its documentation). If no *commandname* is given, display the definitions for all user-defined commands.

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

If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

18.2 User-defined command hooks

You may define *hooks*, which are a special kind of user-defined command. Whenever you run the command '`foo`', if the user-defined command '`hook-foo`' exists, it is executed (with no arguments) before that command.

In addition, a pseudo-command, '`stop`' exists. Defining ('`hook-stop`') makes the associated commands execute every time execution stops in your program: before breakpoint commands are run, displays are printed, or the stack frame is printed.

For example, to ignore `SIGALRM` signals while single-stepping, and treat them normally during normal execution, you could define:

```
define hook-stop
handle SIGALRM nopass
end

define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

You can define a hook for any single-word command in GDB, and not for command aliases; Also you should define a hook for the basic command name, e.g. `backtrace` rather than `bt`. If an error occurs during the execution of your hook, execution of GDB commands stops and GDB issues a prompt (before the command that you actually typed had a chance to run).

If you try to define a hook which does not match any known command, GDB issues a warning from the `define` command.

18.3 Command files

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

When you start GDB, it executes commands from its *init files*. These are files named '`.gdbinit`' on Unix and '`gdb.ini`' on DOS/Windows. During startup, GDB does the following:

1. Reads the init file (if any) in your home directory[1].
2. Processes command line options and operands.
3. Reads the init file (if any) in the current working directory.
4. Reads command files specified by the '`-x`' option.

The init file in your home directory can set options (such as '`set complaints`') that affect subsequent processing of command line options and operands. Init files are not executed if you use the '`-nx`' option (see Section 2.1.2 [Choosing modes], page 13).

It can be useful to create a '`.gdbinit`' file in the directory where you are debugging an application. This file will set the actions that apply for this application.

For example, one might add lines like:

```
dir /usr/src/path/to/source/files
```

to add source directories or:

```
break fatal
```

to set breakpoints on fatal error routines or diagnostic routines.

On some configurations of GDB, the init file is known by a different name (these are typically environments where a specialized form of GDB may need to coexist with other forms, hence a different name for the specialized version's init file). These are the environments with special init file names:

- VxWorks (Wind River Systems real-time OS): '`.vxgdbinit`'
- OS68K (Enea Data Systems real-time OS): '`.os68gdbinit`'
- ES-1800 (Ericsson Telecom AB M68000 emulator): '`.esgdbinit`'

You can also request the execution of a command file with the `source` command:

`source` *`filename`*
: Execute the command file *filename*.

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

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

[1] On DOS/Windows systems, the home directory is the one pointed to by the `HOME` environment variable.

18.4 Commands for controlled output

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

`echo` *`text`*

Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as '`\n`' to print a newline. **No newline is printed unless you specify one.** In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments. To print '` and foo = `', use the command '`echo \ and foo = \ `'.

A backslash at the end of *text* can be used, as in C, to continue the command onto subsequent lines. For example,

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

produces the same output as

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

`output` *`expression`*

Print the value of *expression* and nothing but that value: no newlines, no '`$nn = `'. The value is not entered in the value history either. See Section 8.1 [Expressions], page 63, for more information on expressions.

`output/`*`fmt expression`*

Print the value of *expression* in format *fmt*. You can use the same formats as for `print`. See Section 8.4 [Output formats], page 66, for more information.

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

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

```
printf (string, expressions...);
```

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

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

19 Using GDB under GNU Emacs

A special interface allows you to use GNU Emacs to view (and edit) the source files for the program you are debugging with GDB.

To use this interface, use the command `M-x gdb` in Emacs. Give the executable file you want to debug as an argument. This command starts GDB as a subprocess of Emacs, with input and output through a newly created Emacs buffer.

Using GDB under Emacs is just like using GDB normally except for two things:

- All "terminal" input and output goes through the Emacs buffer.

This applies both to GDB commands and their output, and to the input and output done by the program you are debugging.

This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

All the facilities of Emacs' Shell mode are available for interacting with your program. In particular, you can send signals the usual way—for example, `C-c C-c` for an interrupt, `C-c C-z` for a stop.

- GDB displays source code through Emacs.

Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow ('`=>`') at the left margin of the current line. Emacs uses a separate buffer for source display, and splits the screen to show both your GDB session and the source.

Explicit GDB `list` or search commands still produce output as usual, but you probably have no reason to use them from Emacs.

> *Warning:* If the directory where your program resides is not your current directory, it can be easy to confuse Emacs about the location of the source files, in which case the auxiliary display buffer does not appear to show your source. GDB can find programs by searching your environment's `PATH` variable, so the GDB input and output session proceeds normally; but Emacs does not get enough information back from GDB to locate the source files in this situation. To avoid this problem, either start GDB mode from the directory where your program resides, or specify an absolute file name when prompted for the `M-x gdb` argument.
>
> A similar confusion can result if you use the GDB `file` command to switch to debugging a program in some other location, from an existing GDB buffer in Emacs.

By default, `M-x gdb` calls the program called '`gdb`'. If you need to call GDB by a different name (for example, if you keep several configurations around, with different names) you can set the Emacs variable `gdb-command-name`; for example,

```
(setq gdb-command-name "mygdb")
```

(preceded by `M-:` or `ESC :`, or typed in the `*scratch*` buffer, or in your '`.emacs`' file) makes Emacs call the program named "`mygdb`" instead.

In the GDB I/O buffer, you can use these special Emacs commands in addition to the standard Shell mode commands:

C-h m Describe the features of Emacs' GDB Mode.

M-s Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

M-n Execute to next source line in this function, skipping all function calls, like the GDB `next` command. Then update the display window to show the current file and location.

M-i Execute one instruction, like the GDB `stepi` command; update display window accordingly.

M-x gdb-nexti
Execute to next instruction, using the GDB `nexti` command; update display window accordingly.

C-c C-f Execute until exit from the selected stack frame, like the GDB `finish` command.

M-c Continue execution of your program, like the GDB `continue` command.

Warning: In Emacs v19, this command is *C-c C-p*.

M-u Go up the number of frames indicated by the numeric argument (see section "Numeric Arguments" in *The* GNU *Emacs Manual*), like the GDB `up` command.

Warning: In Emacs v19, this command is *C-c C-u*.

M-d Go down the number of frames indicated by the numeric argument, like the GDB `down` command.

Warning: In Emacs v19, this command is *C-c C-d*.

C-x & Read the number where the cursor is positioned, and insert it at the end of the GDB I/O buffer. For example, if you wish to disassemble code around an address that was displayed earlier, type *disassemble*; then move the cursor to the address display, and pick up the argument for `disassemble` by typing *C-x &*.

You can customize this further by defining elements of the list `gdb-print-command`; once it is defined, you can format or otherwise process numbers picked up by *C-x &* before they are inserted. A numeric argument to *C-x &* indicates that you wish special formatting, and also acts as an index to pick an element of the list. If the list element is a string, the number to be inserted is formatted using the Emacs function `format`; otherwise the number is passed as an argument to the corresponding list element.

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

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command `f` in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

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

20 GDB Annotations

This chapter describes annotations in GDB. Annotations are designed to interface GDB to graphical user interfaces or other similar programs which want to interact with GDB at a relatively high level.

20.1 What is an annotation?

To produce annotations, start GDB with the `--annotate=2` option.

Annotations start with a newline character, two '`control-z`' characters, and the name of the annotation. If there is no additional information associated with this annotation, the name of the annotation is followed immediately by a newline. If there is additional information, the name of the annotation is followed by a space, the additional information, and a newline. The additional information cannot contain newline characters.

Any output not beginning with a newline and two '`control-z`' characters denotes literal output from GDB. Currently there is no need for GDB to output a newline followed by two '`control-z`' characters, but if there was such a need, the annotations could be extended with an '`escape`' annotation which means those three characters as output.

A simple example of starting up GDB with annotations is:

```
$ gdb --annotate=2
GNU GDB 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "sparc-sun-sunos4.1.3"

^Z^Zpre-prompt
(gdb)
^Z^Zprompt
quit

^Z^Zpost-prompt
$
```

Here '`quit`' is input to GDB; the rest is output from GDB. The three lines beginning '`^Z^Z`' (where '`^Z`' denotes a '`control-z`' character) are annotations; the rest is output from GDB.

20.2 The server prefix

To issue a command to GDB without affecting certain aspects of the state which is seen by users, prefix it with '`server `'. This means that this command will not affect the command history, nor will it affect GDB's notion of which command to repeat if ⟨RET⟩ is pressed on a line by itself.

The server prefix does not affect the recording of values into the value history; to print a value without recording it into the value history, use the `output` command instead of the `print` command.

20.3 Values

When a value is printed in various contexts, GDB uses annotations to delimit the value from the surrounding text.

If a value is printed using `print` and added to the value history, the annotation looks like

```
^Z^Zvalue-history-begin history-number value-flags
history-string
^Z^Zvalue-history-value
the-value
^Z^Zvalue-history-end
```

where *history-number* is the number it is getting in the value history, *history-string* is a string, such as '`$5 = `', which introduces the value to the user, *the-value* is the output corresponding to the value itself, and *value-flags* is '`*`' for a value which can be dereferenced and '`-`' for a value which cannot.

If the value is not added to the value history (it is an invalid float or it is printed with the `output` command), the annotation is similar:

```
^Z^Zvalue-begin value-flags
the-value
^Z^Zvalue-end
```

When GDB prints an argument to a function (for example, in the output from the `backtrace` command), it annotates it as follows:

```
^Z^Zarg-begin
argument-name
^Z^Zarg-name-end
separator-string
^Z^Zarg-value value-flags
the-value
^Z^Zarg-end
```

where *argument-name* is the name of the argument, *separator-string* is text which separates the name from the value for the user's benefit (such as '`=`'), and *value-flags* and *the-value* have the same meanings as in a `value-history-begin` annotation.

When printing a structure, GDB annotates it as follows:

```
^Z^Zfield-begin value-flags
field-name
^Z^Zfield-name-end
separator-string
^Z^Zfield-value
the-value
^Z^Zfield-end
```

where *field-name* is the name of the field, *separator-string* is text which separates the name from the value for the user's benefit (such as '`=`'), and *value-flags* and *the-value* have the same meanings as in a `value-history-begin` annotation.

When printing an array, GDB annotates it as follows:

```
^Z^Zarray-section-begin array-index value-flags
```

where *array-index* is the index of the first element being annotated and *value-flags* has the same meaning as in a `value-history-begin` annotation. This is followed by any number of elements, where the element can be either a single element or a repeated element as shown in the examples below:

```
',' whitespace         ; omitted for the first element
the-value
^Z^Zelt
```

```
',' whitespace         ; omitted for the first element
the-value
^Z^Zelt-rep number-of-repititions
repetition-string
^Z^Zelt-rep-end
```

In both cases, *the-value* is the output for the value of the element and *whitespace* can contain spaces, tabs, and newlines. In the repeated case, *number-of-repititons* is the number of consecutive array elements which contain that value, and *repetition-string* is a string which is designed to convey to the user that repitition is being depicted.

Once all the array elements have been output, the array annotation is ended with

```
^Z^Zarray-section-end
```

20.4 Frames

Whenever GDB prints a frame, it annotates it. For example, this applies to frames printed when GDB stops, output from commands such as `backtrace` or `up`, etc.

The frame annotation begins with

```
^Z^Zframe-begin level address
level-string
```

where *level* is the number of the frame (0 is the innermost frame, and other frames have positive numbers), *address* is the address of the code executing in that frame, and *level-string* is a string designed to convey the level to the user. *address* is in the form '`0x`' followed by one or more lowercase hex digits (note that this does not depend on the language). The frame ends with

```
^Z^Zframe-end
```

Between these annotations is the main body of the frame, which can consist of

-
  ```
  ^Z^Zfunction-call
  function-call-string
  ```

  where *function-call-string* is text designed to convey to the user that this frame is associated with a function call made by GDB to a function in the program being debugged.

-
  ```
  ^Z^Zsignal-handler-caller
  signal-handler-caller-string
  ```

  where *signal-handler-caller-string* is text designed to convey to the user that this frame is associated with whatever mechanism is used by this operating system to call a signal handler (it is the frame which calls the signal handler, not the frame for the signal handler itself).

- A normal frame.

  This can optionally (depending on whether this is thought of as interesting information for the user to see) begin with

  ```
  ^Z^Zframe-address
  address
  ^Z^Zframe-address-end
  separator-string
  ```

  where *address* is the address executing in the frame (the same address as in the `frame-begin` annotation, but printed in a form which is intended for user consumption—in particular, the syntax varies depending on the language), and *separator-string* is a string intended to separate this address from what follows for the user's benefit.

  Then comes

  ```
  ^Z^Zframe-function-name
  function-name
  ^Z^Zframe-args
  arguments
  ```

  where *function-name* is the name of the function executing in the frame, or '`??`' if not known, and *arguments* are the arguments to the frame, with parentheses around them (each argument is annotated individually as well, see Section 20.3 [Value Annotations], page 262).

  If source information is available, a reference to it is then printed:

  ```
  ^Z^Zframe-source-begin
  source-intro-string
  ^Z^Zframe-source-file
  filename
  ^Z^Zframe-source-file-end
  :
  ^Z^Zframe-source-line
  line-number
  ^Z^Zframe-source-end
  ```

  where *source-intro-string* separates for the user's benefit the reference from the text which precedes it, *filename* is the name of the source file, and *line-number* is the line number within that file (the first line is line 1).

  If GDB prints some information about where the frame is from (which library, which load segment, etc.; currently only done on the RS/6000), it is annotated with

  ```
  ^Z^Zframe-where
  information
  ```

  Then, if source is to be actually displayed for this frame (for example, this is not true for output from the `backtrace` command), then a `source` annotation (see Section 20.11 [Source Annotations], page 268) is displayed. Unlike most annotations, this is output instead of the normal text which would be output, not in addition.

20.5 Displays

When GDB is told to display something using the `display` command, the results of the display are annotated:

```
^Z^Zdisplay-begin
number
```

```
^Z^Zdisplay-number-end
number-separator
^Z^Zdisplay-format
format
^Z^Zdisplay-expression
expression
^Z^Zdisplay-expression-end
expression-separator
^Z^Zdisplay-value
value
^Z^Zdisplay-end
```

where *number* is the number of the display, *number-separator* is intended to separate the number from what follows for the user, *format* includes information such as the size, format, or other information about how the value is being displayed, *expression* is the expression being displayed, *expression-separator* is intended to separate the expression from the text that follows for the user,and *value* is the actual value being displayed.

20.6 Annotation for GDB input

When GDB prompts for input, it annotates this fact so it is possible to know when to send output, when the output from a given command is over, etc.

Different kinds of input each have a different *input type*. Each input type has three annotations: a `pre-` annotation, which denotes the beginning of any prompt which is being output, a plain annotation, which denotes the end of the prompt, and then a `post-` annotation which denotes the end of any echo which may (or may not) be associated with the input. For example, the `prompt` input type features the following annotations:

```
^Z^Zpre-prompt
^Z^Zprompt
^Z^Zpost-prompt
```

The input types are

`prompt` When GDB is prompting for a command (the main GDB prompt).

`commands` When GDB prompts for a set of commands, like in the `commands` command. The annotations are repeated for each command which is input.

`overload-choice`
When GDB wants the user to select between various overloaded functions.

`query` When GDB wants the user to confirm a potentially dangerous operation.

`prompt-for-continue`
When GDB is asking the user to press return to continue. Note: Don't expect this to work well; instead use `set height 0` to disable prompting. This is because the counting of lines is buggy in the presence of annotations.

20.7 Errors

```
^Z^Zquit
```

This annotation occurs right before GDB responds to an interrupt.

```
^Z^Zerror
```

This annotation occurs right before GDB responds to an error.

Quit and error annotations indicate that any annotations which GDB was in the middle of may end abruptly. For example, if a `value-history-begin` annotation is followed by a `error`, one cannot expect to receive the matching `value-history-end`. One cannot expect not to receive it either; however, an error annotation does not necessarily mean that GDB is immediately returning all the way to the top level.

A quit or error annotation may be preceded by

```
^Z^Zerror-begin
```

Any output between that and the quit or error annotation is the error message.

Warning messages are not yet annotated.

20.8 Information on breakpoints

The output from the `info breakpoints` command is annotated as follows:

```
^Z^Zbreakpoints-headers
header-entry
^Z^Zbreakpoints-table
```

where *header-entry* has the same syntax as an entry (see below) but instead of containing data, it contains strings which are intended to convey the meaning of each field to the user. This is followed by any number of entries. If a field does not apply for this entry, it is omitted. Fields may contain trailing whitespace. Each entry consists of:

```
^Z^Zrecord
^Z^Zfield 0
number
^Z^Zfield 1
type
^Z^Zfield 2
disposition
^Z^Zfield 3
enable
^Z^Zfield 4
address
^Z^Zfield 5
what
^Z^Zfield 6
frame
^Z^Zfield 7
condition
^Z^Zfield 8
ignore-count
^Z^Zfield 9
commands
```

Note that *address* is intended for user consumption—the syntax varies depending on the language.

The output ends with

```
^Z^Zbreakpoints-table-end
```

20.9 Invalidation notices

The following annotations say that certain pieces of state may have changed:

`^Z^Zframes-invalid`
: The frames (for example, output from the `backtrace` command) may have changed.

`^Z^Zbreakpoints-invalid`
: The breakpoints may have changed. For example, the user just added or deleted a breakpoint.

20.10 Running the program

When the program starts executing due to a GDB command such as `step` or `continue`,

```
^Z^Zstarting
```

is output. When the program stops,

```
^Z^Zstopped
```

is output. Before the `stopped` annotation, a variety of annotations describe how the program stopped.

`^Z^Zexited` *`exit-status`*
: The program exited, and *exit-status* is the exit status (zero for successful exit, otherwise nonzero).

`^Z^Zsignalled`
: The program exited with a signal. After the `^Z^Zsignalled`, the annotation continues:

```
intro-text
^Z^Zsignal-name
name
^Z^Zsignal-name-end
middle-text
^Z^Zsignal-string
string
^Z^Zsignal-string-end
end-text
```

where *name* is the name of the signal, such as `SIGILL` or `SIGSEGV`, and *string* is the explanation of the signal, such as `Illegal Instruction` or `Segmentation fault`. *intro-text*, *middle-text*, and *end-text* are for the user's benefit and have no particular format.

`^Z^Zsignal`
: The syntax of this annotation is just like `signalled`, but GDB is just saying that the program received the signal, not that it was terminated with it.

`^Z^Zbreakpoint` *`number`*
: The program hit breakpoint number *number*.

`^Z^Zwatchpoint` *`number`*
: The program hit watchpoint number *number*.

20.11 Displaying source

The following annotation is used instead of displaying source code:

```
^Z^Zsource filename:line:character:middle:addr
```

where *filename* is an absolute file name indicating which source file, *line* is the line number within that file (where 1 is the first line in the file), *character* is the character position within the file (where 0 is the first character in the file) (for most debug formats this will necessarily point to the beginning of a line), *middle* is '`middle`' if *addr* is in the middle of the line, or '`beg`' if *addr* is at the beginning of the line, and *addr* is the address in the target program associated with the source which is being displayed. *addr* is in the form '`0x`' followed by one or more lowercase hex digits (note that this does not depend on the language).

20.12 Annotations We Might Want in the Future

```
- target-invalid
  the target might have changed (registers, heap contents, or
  execution status).  For performance, we might eventually want
  to hit 'registers-invalid' and 'all-registers-invalid' with
  greater precision

- systematic annotation for set/show parameters (including
  invalidation notices).

- similarly, 'info' returns a list of candidates for invalidation
  notices.
```

21 The GDB/MI Interface

Function and purpose

GDB/MI is a line based machine oriented text interface to GDB. It is specifically intended to support the development of systems which use the debugger as just one small component of a larger system.

This chapter is a specification of the GDB/MI interface. It is written in the form of a reference manual.

Notation and terminology

This chapter uses the following notation:

- `|` separates two alternatives.
- `[ *something* ]` indicates that *something* is optional: it may or may not be given.
- `( *group* )*` means that *group* inside the parentheses may repeat zero or more times.
- `( *group* )+` means that *group* inside the parentheses may repeat one or more times.
- `"*string*"` means a literal *string*.

Acknowledgments

In alphabetic order: Andrew Cagney, Fernando Nasser, Stan Shebs and Elena Zannoni.

21.1 GDB/MI Command Syntax

21.1.1 GDB/MI Input syntax

command ↦
: *cli-command* | *mi-command*

cli-command ↦
: [*token*] *cli-command* *nl*, where *cli-command* is any existing GDB CLI command.

mi-command ↦
: [*token*] "-" *operation* (" " *option*)* [" --"] (" " *parameter*)* *nl*

token ↦
: "any sequence of digits"

option ↦
: "-" *parameter* [" " *parameter*]

parameter ↦
: *non-blank-sequence* | *c-string*

operation ↦
    any of the operations described in this chapter

non-blank-sequence ↦
    anything, provided it doesn't contain special characters such as "-", *nl*, """ and of course " "

c-string ↦
    """ *seven-bit-iso-c-string-content* """

nl ↦ CR | CR-LF

Notes:

- The CLI commands are still handled by the MI interpreter; their output is described below.
- The *token*, when present, is passed back when the command finishes.
- Some MI commands accept optional arguments as part of the parameter list. Each option is identified by a leading '-' (dash) and may be followed by an optional argument parameter. Options occur first in the parameter list and can be delimited from normal parameters using '--' (this is useful when some parameters begin with a dash).

Pragmatics:

- We want easy access to the existing CLI syntax (for debugging).
- We want it to be easy to spot a MI operation.

21.1.2 GDB/MI Output syntax

The output from GDB/MI consists of zero or more out-of-band records followed, optionally, by a single result record. This result record is for the most recent command. The sequence of output records is terminated by '(gdb)'.

If an input command was prefixed with a *token* then the corresponding output for that command will also be prefixed by that same *token*.

output ↦
    (*out-of-band-record*)* [*result-record*] "(gdb)" *nl*

result-record ↦
    [*token*] "^" *result-class* ("," *result*)* *nl*

out-of-band-record ↦
    async-record | *stream-record*

async-record ↦
    exec-async-output | *status-async-output* | *notify-async-output*

exec-async-output ↦
    [*token*] "*" *async-output*

status-async-output ↦
    [*token*] "+" *async-output*

notify-async-output ↦
    [*token*] "=" *async-output*

`async-output ↦`
    `async-class ( "," result )* nl`

`result-class ↦`
    `"done" | "running" | "connected" | "error" | "exit"`

`async-class ↦`
    `"stopped" | others` (where *others* will be added depending on the needs—this is still in development).

`result ↦`
    `variable "=" value`

`variable ↦`
    `string`

`value ↦` `const | tuple | list`

`const ↦` `c-string`

`tuple ↦` `"{}" | "{" result ( "," result )* "}"`

`list ↦` `"[]" | "[" value ( "," value )* "]" | "[" result ( "," result )* "]"`

`stream-record ↦`
    `console-stream-output | target-stream-output | log-stream-output`

`console-stream-output ↦`
    `"~" c-string`

`target-stream-output ↦`
    `"@" c-string`

`log-stream-output ↦`
    `"&" c-string`

`nl ↦` `CR | CR-LF`

`token ↦` *any sequence of digits.*

Notes:

- All output sequences end in a single line containing a period.
- The `token` is from the corresponding request. If an execution command is interrupted by the '`-exec-interrupt`' command, the *token* associated with the '`*stopped`' message is the one of the original execution command, not the one of the interrupt command.
- *status-async-output* contains on-going status information about the progress of a slow operation. It can be discarded. All status output is prefixed by '`+`'.
- *exec-async-output* contains asynchronous state change on the target (stopped, started, disappeared). All async output is prefixed by '`*`'.
- *notify-async-output* contains supplementary information that the client should handle (e.g., a new breakpoint information). All notify output is prefixed by '`=`'.
- *console-stream-output* is output that should be displayed as is in the console. It is the textual response to a CLI command. All the console output is prefixed by '`~`'.

- *target-stream-output* is the output produced by the target program. All the target output is prefixed by '`@`'.
- *log-stream-output* is output text coming from GDB's internals, for instance messages that should be displayed as part of an error log. All the log output is prefixed by '`&`'.
- New GDB/MI commands should only output *lists* containing *values*.

See Section 21.3.2 [GDB/MI Stream Records], page 273, for more details about the various output records.

21.1.3 Simple examples of GDB/MI interaction

This subsection presents several simple examples of interaction using the GDB/MI interface. In these examples, '`->`' means that the following line is passed to GDB/MI as input, while '`<-`' means the output received from GDB/MI.

Evaluate expression

Here is an example to evaluate an expression:

```
-> -data-evaluate-expression 2+3
<- (gdb)
<- ^done,value="5"
<- (gdb)
```

and later:

```
<- *stop,reason="stop",address="0x123",source="a.c:123"
<- (gdb)
```

Simple CLI command

Here is an example of a simple CLI command being passed through GDB/MI and on to the CLI.

```
-> print 1+2
<- &"print 1+2\n"
<- ~"$1 = 3\n"
<- ^done
<- (gdb)
```

A bad command

Here is what happens if you pass a bad command:

```
-> -rubbish
<- ^error,msg="Undefined MI command: rubbish"
<- (gdb)
```

21.2 GDB/MI compatibility with CLI

To help users get familiar with GDB CLI, GBB/MI accepts existing CLI commands. As specified by the syntax, such commands can be directly entered into the GDB/MI interface and GDB will respond.

This mechanism is provided as an aid to developers of GDB/MI clients and not as a reliable interface into the CLI. Since the command is being interpreteted in an environment that assumes GDB/MI behaviour, the exact output of such commands is likely to end up being an un-supported hybrid of GDB/MI and CLI output.

21.3 GDB/MI output records

21.3.1 GDB/MI result records

In addition to a number of out-of-band notifications, the response to a GDB/MI command includes one of the following result indications:

`"^done" [ "," *results* ]`
: The synchronous operation was successful, ***results*** are the return values.

`"^running"`
: The asynchronous operation was successfully started. The target is running.

`"^error" "," *c-string*`
: The operation failed. The ***c-string*** contains the corresponding error message.

21.3.2 GDB/MI stream records

GDB internally maintains a number of output streams: the console, the target, and the log. The output intended for each of these streams is funneled through the GDB/MI interface using *stream records*.

Each stream record begins with a unique *prefix character* which identifies its stream (see Section 21.1.2 [GDB/MI Output Syntax], page 270). In addition to the prefix, each stream record contains a ***string-output***. This is either raw text (with an implicit new line) or a quoted C string (which does not contain an implicit newline).

`"~" *string-output*`
: The console output stream contains text that should be displayed in the CLI console window. It contains the textual responses to CLI commands.

`"@" *string-output*`
: The target output stream contains any textual output from the running target.

`"&" *string-output*`
: The log stream contains internal debugging messages being produced by GDB.

21.3.3 GDB/MI out-of-band records

Out-of-band records are used to notify the GDB/MI client of additional changes that have occurred. Those changes can either be a consequence of GDB/MI (e.g., a breakpoint modified) or a result of target activity (e.g., target stopped).

The following is a preliminary list of possible out-of-band records.

`"*" "stop"`

21.4 GDB/MI command description format

The remaining sections describe blocks of commands. Each block of commands is laid out in a fashion similar to this section.

Note the line breaks shown in the examples are here only for readability. They donot appear in the real output. Also note that the commands with a non-available example (N.A.) are not yet implemented.

Motivation

The motivation for this collection of commands.

Introduction

A brief introduction to this collection of commands as a whole.

Commands

For each command in the block, the following is described:

Synopsis

```
     -command args...
```

GDB command

The corresponding GDB CLI command.

Result

Out-of-band

Notes

Example

21.5 GDB/MI breakpoint table commands

This section documents GDB/MI commands for manipulating breakpoints.

The `-break-after` Command

Synopsis

```
 -break-after number count
```

The breakpoint number *number* is not in effect until it has been hit *count* times. To see how this is reflected in the output of the '`-break-list`' command, see the description of the '`-break-list`' command below.

GDB command

The corresponding GDB command is '`ignore`'.

Example

```
(gdb)
-break-insert main
^done,bkpt=number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"(gdb)
-break-after 1 3
~
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0",
ignore="3"}]}
(gdb)
```

The `-break-condition` command

Synopsis

```
-break-condition number expr
```

Breakpoint *number* will stop the program only if the condition in *expr* is true. The condition becomes part of the '`-break-list`' output (see the description of the '`-break-list`' command below).

GDB command

The corresponding GDB command is '`condition`'.

Example

```
(gdb)
-break-condition 1 1
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",cond="1",
times="0",ignore="3"}]}
(gdb)
```

The `-break-delete` command

Synopsis

```
-break-delete ( breakpoint )+
```

Delete the breakpoint(s) whose number(s) are specified in the argument list. This is obviously reflected in the breakpoint list.

GDB command

The corresponding GDB command is '`delete`'.

Example

```
(gdb)
```

```
-break-delete 1
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="0",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[]}
(gdb)
```

The `-break-disable` command

Synopsis

```
 -break-disable ( breakpoint )+
```

Disable the named *breakpoint*(s). The field '`enabled`' in the break list is now set to '`n`' for the named *breakpoint*(s).

GDB command

The corresponding GDB command is '`disable`'.

Example

```
(gdb)
-break-disable 2
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="2",type="breakpoint",disp="keep",enabled="n",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"}]}
(gdb)
```

The `-break-enable` command

Synopsis

```
    -break-enable ( breakpoint )+
```

Enable (previously disabled) *breakpoint*(s).

GDB command

The corresponding GDB command is '`enable`'.

Example

```
(gdb)
-break-enable 2
^done
(gdb)
-break-list
^done,BreakpointTable={nr_rows="1",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="2",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"}]}
(gdb)
```

The `-break-info` Command

Synopsis

```
    -break-info breakpoint
```

Get information about a single breakpoint.

GDB command

The corresponding GDB command is '`info break breakpoint`'.

Example

N.A.

The `-break-insert` command

Synopsis

```
      -break-insert [ -t ] [ -h ] [ -r ]
         [ -c condition ] [ -i ignore-count ]
         [ -p thread ] [ line | addr ]
```

If specified, *line*, can be one of:

- function
- filename:linenum
- filename:function
- *address

The possible optional parameters of this command are:

'`-t`' Insert a temporary breakpoint.

'`-h`' Insert a hardware breakpoint.

'`-c` *condition*'
Make the breakpoint conditional on *condition*.

'`-i` *ignore-count*'
Initialize the *ignore-count*.

'`-r`' Insert a regular breakpoint in all the functions whose names match the given regular expression. Other flags are not applicable to regular expression.

Result

The result is in the form:

```
    ^done,bkptno="number",func="funcname",
     file="filename",line="lineno"
```

where *number* is the GDB number for this breakpoint, *funcname* is the name of the function where the breakpoint was inserted, *filename* is the name of the source file which contains this function, and *lineno* is the source line number within that file.

Note: this format is open to change.

GDB command

The corresponding GDB commands are '`break`', '`tbreak`', '`hbreak`', '`thbreak`', and '`rbreak`'.

Example

```
(gdb)
-break-insert main
^done,bkpt=number="1",type="breakpoint",disp="keep",enabled="y",addr="0x0001072c",
file="recursive2.c",line="4",times="0"
(gdb)
-break-insert -t foo
```

```
^done,bkpt=number="2",type="breakpoint",disp="keep",enabled="y",addr="0x00010774",
file="recursive2.c",line="11",times="0"(gdb)
-break-list
^done,BreakpointTable={nr_rows="2",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x0001072c", func="main",file="recursive2.c",line="4",times="0"},
bkpt={number="2",type="breakpoint",disp="del",enabled="y",
addr="0x00010774",func="foo",file="recursive2.c",line="11",times="0"}]}
(gdb)
-break-insert -r foo.*
~int foo(int, int);
^done,bkpt={number="3",addr="0x00010774",file="recursive2.c",line="11"}
(gdb)
```

The `-break-list` command

Synopsis

```
 -break-list
```

Displays the list of inserted breakpoints, showing the following fields:

‘`Number`’	number of the breakpoint
‘`Type`’	type of the breakpoint: ‘`breakpoint`’ or ‘`watchpoint`’
‘`Disposition`’	should the breakpoint be deleted or disabled when it is hit: ‘`keep`’ or ‘`nokeep`’
‘`Enabled`’	is the breakpoint enabled or no: ‘`y`’ or ‘`n`’
‘`Address`’	memory location at which the breakpoint is set
‘`What`’	logical location of the breakpoint, expressed by function name, file name, line number
‘`Times`’	number of times the breakpoint has been hit

If there are no breakpoints or watchpoints, the `BreakpointTable` `body` field is an empty list.

GDB command

The corresponding GDB command is ‘`info break`’.

Example

```
(gdb)
-break-list
```

```
^done,BreakpointTable={nr_rows="2",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[bkpt={number="1",type="breakpoint",disp="keep",enabled="y",
addr="0x000100d0",func="main",file="hello.c",line="5",times="0"},
bkpt={number="2",type="breakpoint",disp="keep",enabled="y",
addr="0x00010114",func="foo",file="hello.c",line="13",times="0"}]}
(gdb)
```

Here's an example of the result when there are no breakpoints:

```
(gdb)
-break-list
^done,BreakpointTable={nr_rows="0",nr_cols="6",
hdr=[{width="3",alignment="-1",col_name="number",colhdr="Num"},
{width="14",alignment="-1",col_name="type",colhdr="Type"},
{width="4",alignment="-1",col_name="disp",colhdr="Disp"},
{width="3",alignment="-1",col_name="enabled",colhdr="Enb"},
{width="10",alignment="-1",col_name="addr",colhdr="Address"},
{width="40",alignment="2",col_name="what",colhdr="What"}],
body=[]}
(gdb)
```

The `-break-watch` command

Synopsis

```
 -break-watch [ -a | -r ]
```

Create a watchpoint. With the '`-a`' option it will create an *access* watchpoint, i.e. a watchpoint that triggers either on a read from or on a write to the memory location. With the '`-r`' option, the watchpoint created is a *read* watchpoint, i.e. it will trigger only when the memory location is accessed for reading. Without either of the options, the watchpoint created is a regular watchpoint, i.e. it will trigger when the memory location is accessed for writing. See ⟨undefined⟩ [Setting watchpoints], page ⟨undefined⟩.

Note that '`-break-list`' will report a single list of watchpoints and breakpoints inserted.

GDB command

The corresponding GDB commands are '`watch`', '`awatch`', and '`rwatch`'.

Example

Setting a watchpoint on a variable in the `main` function:

```
(gdb)
-break-watch i
^done,wpt=number="2",exp="i"
(gdb)
-exec-continue
```

```
^running
(gdb)
*stopped,reason="watchpoint-trigger",wpt=number="2",exp="i",value=old="0",new="7"
,thread-id="1",frame=addr="0x000029c4",func="main",args=[],file="hello.c",line="8"
(gdb)
```

Setting a watchpoint on a variable local to a function. GDB will stop the program execution twice: first for the variable changing value, then for the watchpoint going out of scope.

```
(gdb)
-break-watch j
^done,wpt=number="2",exp="j"
(gdb)
-exec-continue
^running
(gdb)
*stopped,reason="watchpoint-trigger",wpt=number="2",exp="j",value=old="0",new="17",
thread-id="1",frame=addr="0x000029bc",func="call",args=[],file="hello.c",line="10"
(gdb)
-exec-continue
^running
(gdb)
*stopped,reason="watchpoint-scope",wpnum="2",thread-id="1",frame=addr="0x000029ec",
func="main",args=[],file="hello.c",line="18"
(gdb)
```

Listing breakpoints and watchpoints, at different points in the program execution. Note that once the watchpoint goes out of scope, it is deleted.

```
-break-watch j
^done,wpt=number="2",exp="j"
(gdb)
-break-list
^done,BreakpointTable=nr_rows="2",nr_cols="6",hdr=[width="3",
alignment="-1",col_name="number",colhdr="Num",width="14",alignment="-1",
col_name="type",colhdr"Type",width="4",alignment="-1",col_name="disp",
colhdr="Disp",width="3",alinment="-1",col_name="enabled",colhdr="Enb",
width="10",alignment="-1",col_name"addr",colhdr="Address",
width="40",alignment="2",col_name="what",colhdr="What],body=[bkpt=number="1",
type="breakpoint",disp="keep",enabled="y",addr="0x00029b4",func="call",
file="hello.c",line="9",times="1",bkpt=number="2",type="wathpoint",disp="keep",
enabled="y",addr="",what="j",times="0"]
(gdb)
-exec-continue
^running
(gdb)
*stopped,reason="watchpoint-trigger",wpt=number="2",exp="j",value=old="0",ne="17",
thread-id="1",frame=addr="0x000029bc",func="call",args=[],file="hello.c,line="10"
(gdb)
-break-list
^done,BreakpointTable=nr_rows="2",nr_cols="6",hdr=[width="3",alignment="-1",
col_name="number",colhdr="Num",width="14",alignment="-1",col_name="type",colhdr"Type",
width="4",alignment="-1",col_name="disp",colhdr="Disp",width="3",alinment="-1",
col_name="enabled",colhdr="Enb",width="10",alignment="-1",col_name"addr",colhdr="Address",
width="40",alignment="2",col_name="what",colhdr="What],body=[bkpt=number="1",
type="breakpoint",disp="keep",enabled="y",addr="0x00029b4",func="call",file="hello.c",
line="9",times="1",bkpt=number="2",type="wathpoint",disp="keep",enabled="y",addr="",
what="j",times="1"]
(gdb)
```

```
-exec-continue
^running
(gdb)
*stopped,reason="watchpoint-scope",wpnum="2",thread-id="1",frame=addr="0x000029ec",
func="main",args=[],file="hello.c",line="18"
(gdb)
-break-list
^done,BreakpointTable=nr_rows="1",nr_cols="6",hdr=[width="3",alignment="-1",
col_name="number",colhdr="Num",width="14",alignment="-1",col_name="type",colhdr="Type",
width="4",alignment="-1",col_name="disp",colhdr="Disp",width="3",alignment="-1",
col_name="enabled",colhdr="Enb",width="10",alignment="-1",col_name="addr",
colhdr="Address",width="40",alignment="2",col_name="what",colhdr="What"],
body=[bkpt=number="1",type="breakpoint",disp="keep",enabled="y",addr="0x000029b4",
func="call",file="hello.c",line="9",times="1"]
(gdb)
```

21.6 GDB/MI Data manipulation

This section describes the GDB/MI commands that manipulate data: examine memory and registers, evaluate expressions, etc.

The `-data-disassemble` command

Synopsis

```
 -data-disassemble
    [ -s start-addr -e end-addr ]
  | [ -f filename -l linenum [ -n lines ] ]
  -- mode
```

Where:

‘*start-addr*’
: is the beginning address (or `$pc`)

‘*end-addr*’
: is the end address

‘*filename*’
: is the name of the file to disassemble

‘*linenum*’ is the line number to disassemble around

‘*lines*’ is the the number of disassembly lines to be produced. If it is -1, the whole function will be disassembled, in case no *end-addr* is specified. If *end-addr* is specified as a non-zero value, and *lines* is lower than the number of disassembly lines between *start-addr* and *end-addr*, only *lines* lines are displayed; if *lines* is higher than the number of lines between *start-addr* and *end-addr*, only the lines up to *end-addr* are displayed.

‘*mode*’ is either 0 (meaning only disassembly) or 1 (meaning mixed source and disassembly).

Result

The output for each instruction is composed of four fields:

- Address
- Func-name
- Offset
- Instruction

Note that whatever included in the instruction field, is not manipulated directly by GDB/MI, i.e. it is not possible to adjust its format.

GDB command

There is no direct mapping from this command to the CLI.

Example

Disassemble from the current value of `$pc` to `$pc + 20`:

```
(gdb)
-data-disassemble -s $pc -e "$pc + 20" -- 0
^done,
asm_insns=[
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"},
{address="0x000107c8",func-name="main",offset="12",
inst="or  %o2, 0x140, %o1\t! 0x11940 <_lib_version+8>"},
{address="0x000107cc",func-name="main",offset="16",
inst="sethi  %hi(0x11800), %o2"},
{address="0x000107d0",func-name="main",offset="20",
inst="or  %o2, 0x168, %o4\t! 0x11968 <_lib_version+48>"}]
(gdb)
```

Disassemble the whole `main` function. Line 32 is part of `main`.

```
-data-disassemble -f basics.c -l 32 -- 0
^done,asm_insns=[
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"},
{address="0x000107c0",func-name="main",offset="4",
inst="mov   2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi %hi(0x11800), %o2"},
[...]
{address="0x0001081c",func-name="main",offset="96",inst="ret "},
{address="0x00010820",func-name="main",offset="100",inst="restore "}]
(gdb)
```

Disassemble 3 instructions from the start of `main`:

```
(gdb)
-data-disassemble -f basics.c -l 32 -n 3 -- 0
^done,asm_insns=[
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"},
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"}]
(gdb)
```

Disassemble 3 instructions from the start of `main` in mixed mode:

```
(gdb)
-data-disassemble -f basics.c -l 32 -n 3 -- 1
^done,asm_insns=[
src_and_asm_line={line="31",
file="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb/ \
  testsuite/gdb.mi/basics.c",line_asm_insn=[
{address="0x000107bc",func-name="main",offset="0",
inst="save  %sp, -112, %sp"}]},
src_and_asm_line={line="32",
file="/kwikemart/marge/ezannoni/flathead-dev/devo/gdb/ \
  testsuite/gdb.mi/basics.c",line_asm_insn=[
{address="0x000107c0",func-name="main",offset="4",
inst="mov  2, %o0"},
{address="0x000107c4",func-name="main",offset="8",
inst="sethi  %hi(0x11800), %o2"}]}]
(gdb)
```

The `-data-evaluate-expression` command

Synopsis

```
 -data-evaluate-expression expr
```

Evaluate *expr* as an expression. The expression could contain an inferior function call. The function call will execute synchronously. If the expression contains spaces, it must be enclosed in double quotes.

GDB command

The corresponding GDB commands are '`print`', '`output`', and '`call`'. In `gdbtk` only, there's a corresponding '`gdb_eval`' command.

Example

In the following example, the numbers that precede the commands are the *tokens* described in Section 21.1 [GDB/MI Command Syntax], page 269. Notice how GDB/MI returns the same tokens in its output.

```
211-data-evaluate-expression A
211^done,value="1"
(gdb)
311-data-evaluate-expression &A
311^done,value="0xefffeb7c"
(gdb)
411-data-evaluate-expression A+3
```

```
411^done,value="4"
(gdb)
511-data-evaluate-expression "A + 3"
511^done,value="4"
(gdb)
```

The `-data-list-changed-registers` Command

Synopsis

```
-data-list-changed-registers
```

Display a list of the registers that have changed.

GDB command

GDB doesnot have a direct analog for this command; `gdbtk` has the corresponding command '`gdb_changed_register_list`'.

Example

On a PPC MBX board:

```
(gdb)
-exec-continue
^running

(gdb)
*stopped,reason="breakpoint-hit",bkptno="1",frame={func="main",
args=[],file="try.c",line="5"}
(gdb)
-data-list-changed-registers
^done,changed-registers=["0","1","2","4","5","6","7","8","9",
"10","11","13","14","15","16","17","18","19","20","21","22","23",
"24","25","26","27","28","30","31","64","65","66","67","69"]
(gdb)
```

The `-data-list-register-names` command

Synopsis

```
-data-list-register-names [ ( regno )+ ]
```

Show a list of register names for the current target. If no arguments are given, it shows a list of the names of all the registers. If integer numbers are given as arguments, it will print a list of the names of the registers corresponding to the arguments. To ensure consistency between a register name and its number, the output list may include empty register names.

GDB command

GDB does not have a command which corresponds to '`-data-list-register-names`'. In `gdbtk` there is a corresponding command '`gdb_regnames`'.

Example

For the PPC MBX board:

```
(gdb)
-data-list-register-names
^done,register-names=["r0","r1","r2","r3","r4","r5","r6","r7",
"r8","r9","r10","r11","r12","r13","r14","r15","r16","r17","r18",
"r19","r20","r21","r22","r23","r24","r25","r26","r27","r28","r29",
"r30","r31","f0","f1","f2","f3","f4","f5","f6","f7","f8","f9",
"f10","f11","f12","f13","f14","f15","f16","f17","f18","f19","f20",
"f21","f22","f23","f24","f25","f26","f27","f28","f29","f30","f31",
"", "pc","ps","cr","lr","ctr","xer"]
(gdb)
-data-list-register-names 1 2 3
^done,register-names=["r1","r2","r3"]
(gdb)
```

The `-data-list-register-values` command

Synopsis

```
 -data-list-register-values fmt [ ( regno )*]
```

Display the registers contents. *fmt* is the format according to which the registers' contents are to be returned, followed by an optional list of numbers specifying the registers to display. A missing list of numbers indicates that the contents of all the registers must be returned.

Allowed formats for *fmt* are:

`x`	Hexadecimal
`o`	Octal
`t`	Binary
`d`	Decimal
`r`	Raw
`N`	Natural

GDB command

The corresponding GDB commands are '`info reg`', '`info all-reg`', and (in `gdbtk`) '`gdb_fetch_registers`'.

Example

For a PPC MBX board (note: line breaks are for readability only, they donot appear in the actual output):

```
(gdb)
-data-list-register-values r 64 65
^done,register-values=[{number="64",value="0xfe00a300"},
{number="65",value="0x00029002"}]
(gdb)
-data-list-register-values x
^done,register-values=[{number="0",value="0xfe0043c8"},
{number="1",value="0x3fff88"},{number="2",value="0xfffffffe"},
{number="3",value="0x0"},{number="4",value="0xa"},
{number="5",value="0x3fff68"},{number="6",value="0x3fff58"},
{number="7",value="0xfe011e98"},{number="8",value="0x2"},
{number="9",value="0xfa202820"},{number="10",value="0xfa202808"},
{number="11",value="0x1"},{number="12",value="0x0"},
{number="13",value="0x4544"},{number="14",value="0xffdfffff"},
{number="15",value="0xffffffff"},{number="16",value="0xfffffeff"},
{number="17",value="0xefffffed"},{number="18",value="0xfffffffe"},
{number="19",value="0xffffffff"},{number="20",value="0xffffffff"},
{number="21",value="0xffffffff"},{number="22",value="0xfffffff7"},
{number="23",value="0xffffffff"},{number="24",value="0xffffffff"},
{number="25",value="0xffffffff"},{number="26",value="0xfffffffb"},
{number="27",value="0xffffffff"},{number="28",value="0xf7bfffff"},
{number="29",value="0x0"},{number="30",value="0xfe010000"},
{number="31",value="0x0"},{number="32",value="0x0"},
{number="33",value="0x0"},{number="34",value="0x0"},
{number="35",value="0x0"},{number="36",value="0x0"},
{number="37",value="0x0"},{number="38",value="0x0"},
{number="39",value="0x0"},{number="40",value="0x0"},
{number="41",value="0x0"},{number="42",value="0x0"},
{number="43",value="0x0"},{number="44",value="0x0"},
{number="45",value="0x0"},{number="46",value="0x0"},
{number="47",value="0x0"},{number="48",value="0x0"},
{number="49",value="0x0"},{number="50",value="0x0"},
{number="51",value="0x0"},{number="52",value="0x0"},
{number="53",value="0x0"},{number="54",value="0x0"},
{number="55",value="0x0"},{number="56",value="0x0"},
{number="57",value="0x0"},{number="58",value="0x0"},
{number="59",value="0x0"},{number="60",value="0x0"},
{number="61",value="0x0"},{number="62",value="0x0"},
{number="63",value="0x0"},{number="64",value="0xfe00a300"},
{number="65",value="0x29002"},{number="66",value="0x202f04b5"},
{number="67",value="0xfe0043b0"},{number="68",value="0xfe00b3e4"},
{number="69",value="0x20002b03"}]
(gdb)
```

The `-data-read-memory` command

Synopsis

```
 -data-read-memory [ -o byte-offset ]
   address word-format word-size
```

```
         nr-rows nr-cols [ aschar ]
```

where:

‘*address*’ An expression specifying the address of the first memory word to be read. Complex expressions containing embedded white space should be quoted using the C convention.

‘*word-format*’
The format to be used to print the memory words. The notation is the same as for GDB `print` command (see Section 8.4 [Output formats], page 66).

‘*word-size*’
The size of each memory word in bytes.

‘*nr-rows*’ The number of rows in the output table.

‘*nr-cols*’ The number of columns in the output table.

‘*aschar*’ If present, indicates that each row should include an ASCII dump. The value of *aschar* is used as a padding character when a byte is not a member of the printable ASCII character set (printable ASCII characters are those whose code is between 32 and 126, inclusively).

‘*byte-offset*’
An offset to add to the *address* before fetching memory.

This command displays memory contents as a table of *nr-rows* by *nr-cols* words, each word being *word-size* bytes. In total, `nr-rows * nr-cols * word-size` bytes are read (returned as ‘`total-bytes`’). Should less than the requested number of bytes be returned by the target, the missing words are identified using ‘`N/A`’. The number of bytes read from the target is returned in ‘`nr-bytes`’ and the starting address used to read memory in ‘`addr`’.

The address of the next/previous row or page is available in ‘`next-row`’ and ‘`prev-row`’, ‘`next-page`’ and ‘`prev-page`’.

GDB command

The corresponding GDB command is ‘`x`’. `gdbtk` has ‘`gdb_get_mem`’ memory read command.

Example

Read six bytes of memory starting at `bytes+6` but then offset by `-6` bytes. Format as three rows of two columns. One byte per word. Display each word in hex.

```
(gdb)
9-data-read-memory -o -6 -- bytes+6 x 1 3 2
9^done,addr="0x00001390",nr-bytes="6",total-bytes="6",
next-row="0x00001396",prev-row="0x0000138e",next-page="0x00001396",
prev-page="0x0000138a",memory=[
{addr="0x00001390",data=["0x00","0x01"]},
{addr="0x00001392",data=["0x02","0x03"]},
{addr="0x00001394",data=["0x04","0x05"]}]
(gdb)
```

Read two bytes of memory starting at address `shorts + 64` and display as a single word formatted in decimal.

```
(gdb)
5-data-read-memory shorts+64 d 2 1 1
5^done,addr="0x00001510",nr-bytes="2",total-bytes="2",
next-row="0x00001512",prev-row="0x0000150e",
next-page="0x00001512",prev-page="0x0000150e",memory=[
{addr="0x00001510",data=["128"]}]
(gdb)
```

Read thirty two bytes of memory starting at `bytes+16` and format as eight rows of four columns. Include a string encoding with '`x`' used as the non-printable character.

```
(gdb)
4-data-read-memory bytes+16 x 1 8 4 x
4^done,addr="0x000013a0",nr-bytes="32",total-bytes="32",
next-row="0x000013c0",prev-row="0x0000139c",
next-page="0x000013c0",prev-page="0x00001380",memory=[
{addr="0x000013a0",data=["0x10","0x11","0x12","0x13"],ascii="xxxx"},
{addr="0x000013a4",data=["0x14","0x15","0x16","0x17"],ascii="xxxx"},
{addr="0x000013a8",data=["0x18","0x19","0x1a","0x1b"],ascii="xxxx"},
{addr="0x000013ac",data=["0x1c","0x1d","0x1e","0x1f"],ascii="xxxx"},
{addr="0x000013b0",data=["0x20","0x21","0x22","0x23"],ascii=" !\"#"},
{addr="0x000013b4",data=["0x24","0x25","0x26","0x27"],ascii="$%&'"},
{addr="0x000013b8",data=["0x28","0x29","0x2a","0x2b"],ascii="()*+"},
{addr="0x000013bc",data=["0x2c","0x2d","0x2e","0x2f"],ascii=",-./"}]
(gdb)
```

The `-display-delete` command

Synopsis

```
    -display-delete number
```

Delete the display *number*.

GDB command

The corresponding GDB command is '`delete display`'.

Example

N.A.

The `-display-disable` Command

Synopsis

```
    -display-disable number
```

Disable display *number*.

GDB command

The corresponding GDB command is '`disable display`'.

Example

N.A.

The `-display-enable` command

Synopsis

```
    -display-enable number
```

Enable display *number*.

GDB command

The corresponding GDB command is '`enable display`'.

Example

N.A.

The `-display-insert` Command

Synopsis

```
    -display-insert expression
```

Display *expression* every time the program stops.

GDB command

The corresponding GDB command is '`display`'.

Example

N.A.

The `-display-list` command

Synopsis

```
    -display-list
```

List the displays. Do not show the current values.

GDB command

The corresponding GDB command is '`info display`'.

Example

N.A.

The `-environment-cd` command

Synopsis

```
    -environment-cd pathdir
```

Set the GDB working directory.

GDB command

The corresponding GDB command is '`cd`'.

Example

```
(gdb)
-environment-cd /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

The `-environment-directory` command

Synopsis

```
    -environment-directory pathdir
```

Add directory *pathdir* to the beginning of search path for source files.

GDB command

The corresponding GDB command is '`dir`'.

Example

```
(gdb)
-environment-directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb
^done
(gdb)
```

The `-environment-path` command

Synopsis

```
 -environment-path ( pathdir )+
```

Add directories *pathdir* to beginning of search path for object files.

GDB command

The corresponding GDB command is '`path`'.

Example

```
(gdb)
-environment-path /kwikemart/marge/ezannoni/flathead-dev/ppc-eabi/gdb
^done
(gdb)
```

The `-environment-pwd` command

Synopsis

```
 -environment-pwd
```

Show the current working directory.

GDB command

The corresponding GDB command is '`pwd`'.

Example

```
(gdb)
-environment-pwd
~Working directory /kwikemart/marge/ezannoni/flathead-dev/devo/gdb.
^done
(gdb)
```

21.7 GDB/MI program control

Program termination

As a result of execution, the inferior program can run to completion, if it doesnot encounter any breakpoints. In this case the output will include an exit code, if the program has exited exceptionally.

Examples

Program exited normally:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited-normally"
(gdb)
```

Program exited exceptionally:

```
(gdb)
-exec-run
^running
(gdb)
x = 55
*stopped,reason="exited",exit-code="01"
(gdb)
```

Another way the program can terminate is if it receives a signal such as `SIGINT`. In this case, GDB/MI displays this:

```
(gdb)
*stopped,reason="exited-signalled",signal-name="SIGINT",
signal-meaning="Interrupt"
```

The `-exec-abort` command

Synopsis

```
 -exec-abort
```

Kill the inferior running program.

GDB command

The corresponding GDB command is '`kill`'.

Example

N.A.

The `-exec-arguments` command

Synopsis

```
 -exec-arguments args
```

Set the inferior program arguments, to be used in the next '`-exec-run`'.

GDB command

The corresponding GDB command is '`set args`'.

Example

Donot have one around.

The `-exec-continue` command

Synopsis

```
 -exec-continue
```

Asynchronous command. Resumes the execution of the inferior program until a breakpoint is encountered, or until the inferior exits.

GDB command

The corresponding GDB is '`continue`'.

Example

```
-exec-continue
^running
(gdb)
@Hello world
*stopped,reason="breakpoint-hit",bkptno="2",thread-id="1",frame=addr="0x000029d8",
func="foo",args=[],file="hello.c",line="16"file="hello.c",line="13"}
(gdb)
```

The `-exec-finish` command

Synopsis

```
 -exec-finish
```

Asynchronous command. Resumes the execution of the inferior program until the current function is exited. Displays the results returned by the function.

GDB command

The corresponding GDB command is ‘`finish`’.

Example

Function returning `void`.

```
-exec-finish
^running
(gdb)
@hello from foo
*stopped,reason="function-finished",thread-id="1",frame=addr="0x000029ec",
func="main",args=[],file="hello.c",line="7file="hello.c",line="7"}
(gdb)
```

Function returning other than `void`. The name of the internal GDB variable storing the result is printed, together with the value itself.

```
-exec-finish
^running
(gdb)
*stopped,reason="function-finished",thread-id="1",
frame=addr="0x000107b0",func="foo",
args=[name="a"],name="b",
file="recursive2.c",line="14"},
gdb-result-var="$1",return-value="0"
(gdb)
```

The `-exec-interrupt` command

Synopsis

```
 -exec-interrupt
```

Asynchronous command. Interrupts the background execution of the target. Note how the token associated with the stop message is the one for the execution command that has been interrupted. The token for the interrupt itself only appears in the ‘`^done`’ output. If the user is trying to interrupt a non-running program, an error message will be printed.

GDB command

The corresponding GDB command is ‘`interrupt`’.

Example

```
(gdb)
111-exec-continue
111^running
```

```
(gdb)
222-exec-interrupt
222^done
(gdb)
111*stopped,signal-name="SIGINT",signal-meaning="Interrupt",
frame={addr="0x00010140",func="foo",args=[],file="try.c",line="13"}
(gdb)

(gdb)
-exec-interrupt
^error,msg="mi_cmd_exec_interrupt: Inferior not executing."
(gdb)
```

The `-exec-next` command

Synopsis

```
 -exec-next
```

Asynchronous command. Resumes execution of the inferior program, stopping when the beginning of the next source line is reached.

GDB command

The corresponding GDB command is ‘`next`’.

Example

```
-exec-next
^running
(gdb)
*stopped,reason="end-stepping-range",thread-id="1",frame=addr="0x00002a10",
func="main",args=[],file="hello.c",line="24"(gdb)
```

The `-exec-next-instruction` command

Synopsis

```
 -exec-next-instruction
```

Asynchronous command. Executes one machine instruction. If the instruction is a function call continues until the function returns. If the program stops at an instruction in the middle of a source line, the address will be printed as well.

GDB command

The corresponding GDB command is ‘`nexti`’.

Example

```
(gdb)
-exec-next-instruction
^running

(gdb)
*stopped,reason="end-stepping-range",thread-
id="1",frame=addr="0x00002a14",func="main",args=[],file="hello.c",line="24"
(gdb)
```

The -exec-return command

Synopsis

```
 -exec-return
```

Makes current function return immediately. Doesn't execute the inferior. Displays the new current frame.

GDB command

The corresponding GDB command is '`return`'.

Example

```
(gdb)
-break-insert call1
^done,bkpt=number="1",type="breakpoint",disp="keep",enabled="y",addr="0x000029ac",
func="call1",file="hello.c",line="9",times="0"
(gdb)
-exec-run
^running
(gdb)
~"3"
*stopped,reason="breakpoint-hit",bkptno="1",thread-id="1",frame=addr="0x000029ac",
func="call1",args=[name="a"],file="hello.c",line="9"
(gdb)
-exec-return
~"2"
~"3"
^done,frame=level="0 ",addr="0x000029e8",func="call",args=[name="a",name="b"],
file="hello.c",line="17"
(gdb)
```

The -exec-run command

Synopsis

```
 -exec-run
```

Asynchronous command. Starts execution of the inferior from the beginning. The inferior executes until either a breakpoint is encountered or the program exits.

GDB command

The corresponding GDB command is '`run`'.

Example

```
(gdb)
-break-insert main
^done,bkpt=number="1",type="breakpoint",disp="keep",enabled="y",addr="0x00002a08",
func="main",file="hello.c",line="23",times="0"
(gdb)
-exec-run
^running
(gdb)
*stopped,reason="breakpoint-hit",bkptno="1",thread-id="1",frame=addr="0x00002a08",
func="main",args=[],file="hello.c",line="23"
(gdb)
```

The `-exec-show-arguments` command

Synopsis

```
 -exec-show-arguments
```

Print the arguments of the program.

GDB command

The corresponding GDB command is '`show args`'.

Example

N.A.

The `-exec-step` Command

Synopsis

```
 -exec-step
```

Asynchronous command. Resumes execution of the inferior program, stopping when the beginning of the next source line is reached, if the next source line is not a function call. If it is, stop at the first instruction of the called function.

GDB command

The corresponding GDB command is '`step`'.

Example

Stepping into a function:

```
-exec-step
^running
(gdb)
~"2"
~"3"
*stopped,reason="end-stepping-range",thread-id="1",frame=addr="0x000029d0",
func="call",args=[name="a",name="b"],file="hello.c",line="15"
(gdb)
```

Regular stepping:

```
-exec-step
^running
(gdb)
~"2"
~"3"
*stopped,reason="end-stepping-range",thread-id="1",frame=addr="0x000029d8",
func="call",args=[name="a",name="b"],file="hello.c",line="16"
(gdb)
```

The `-exec-step-instruction` command

Synopsis

```
 -exec-step-instruction
```

Asynchronous command. Resumes the inferior which executes one machine instruction. The output, once GDB has stopped, will vary depending on whether we have stopped in the middle of a source line or not. In the former case, the address at which the program stopped will be printed as well.

GDB command

The corresponding GDB command is '`stepi`'.

Example

```
(gdb)
-exec-step-instruction
^running
(gdb)
~"2"
~"3"
*stopped,reason="end-stepping-range",thread-id="1",frame=addr="0x000029dc",
func="call",args=[name="a",name="b"],file="hello.c",line="16"
```

```
(gdb)

-exec-step-instruction
^running

(gdb)
*stopped,reason="end-stepping-range",
frame={addr="0x000100f4",func="foo",args=[],file="try.c",line="10"}
(gdb)
```

The `-exec-until` command

Synopsis

```
 -exec-until [ location ]
```

Asynchronous command. Executes the inferior until the *location* specified in the argument is reached. If there is no argument, the inferior executes until a source line greater than the current one is reached. The reason for stopping in this case will be '`location-reached`'.

GDB command

The corresponding GDB command is '`until`'.

Example

```
(gdb)
-exec-until recursive2.c:6
^running
(gdb)
x = 55
*stopped,reason="location-reached",thread-id="1",frame=addr="0x00002a24",func="main",args=[],
file="recursive2.c",line="6"
(gdb)
```

The `-file-exec-and-symbols` command

Synopsis

```
 -file-exec-and-symbols file
```

Specify the executable file to be debugged. This file is the one from which the symbol table is also read. If no file is specified, the command clears the executable and symbol information. If breakpoints are set when using this command with no arguments, GDB will produce error messages. Otherwise, no output is produced, except a completion notification.

GDB command

The corresponding GDB command is '`file`'.

Example

```
(gdb)
-file-exec-and-symbols /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

The `-file-exec-file` command

Synopsis

```
 -file-exec-file file
```

Specify the executable file to be debugged. Unlike '`-file-exec-and-symbols`', the symbol table is *not* read from this file. If used without argument, GDB clears the information about the executable file. No output is produced, except a completion notification.

GDB command

The corresponding GDB command is '`exec-file`'.

Example

```
(gdb)
-file-exec-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
^done
(gdb)
```

The `-file-list-exec-sections` command

Synopsis

```
 -file-list-exec-sections
```

List the sections of the current executable file.

GDB command

The GDB command '`info file`' shows, among the rest, the same information as this command. `gdbtk` has a corresponding command '`gdb_load_info`'.

Example

N.A.

The `-file-list-exec-source-files` command

Synopsis

```
-file-list-exec-source-files
```

List the source files for the current executable.

GDB command

There's no GDB command which directly corresponds to this one. `gdbtk` has an analogous command '`gdb_listfiles`'.

Example

N.A.

The `-file-list-shared-libraries` Command

Synopsis

```
-file-list-shared-libraries
```

List the shared libraries in the program.

GDB command

The corresponding GDB command is '`info shared`'.

Example

N.A.

The `-file-list-symbol-files` command

Synopsis

```
-file-list-symbol-files
```

List symbol files.

GDB command

The corresponding GDB command is '`info file`' (part of it).

Example

N.A.

The `-file-symbol-file` Command

Synopsis

```
     -file-symbol-file file
```

Read symbol table info from the specified *file* argument. When used without arguments, clears GDB's symbol table info. No output is produced, except for a completion notification.

GDB command

The corresponding GDB command is '`symbol-file`'.

Example

```
    (gdb)
    -file-symbol-file /kwikemart/marge/ezannoni/TRUNK/mbx/hello.mbx
    ^done
    (gdb)
```

21.8 Miscellaneous GDB commands in GDB/MI

The `-gdb-exit` Command

Synopsis

```
     -gdb-exit
```

Exit GDB immediately.

GDB command

Approximately corresponds to '`quit`'.

Example

```
    (gdb)
    -gdb-exit
```

The `-gdb-set` command

Synopsis

```
     -gdb-set
```

Set an internal GDB variable.

GDB command

The corresponding GDB command is '`set`'.

Example

```
(gdb)
-gdb-set $foo=3
^done
(gdb)
```

The `-gdb-show` command

Synopsis

```
 -gdb-show
```

Show the current value of a GDB variable.

GDB command

The corresponding GDB command is '`show`'.

Example

```
(gdb)
-gdb-show annotate
^done,value="0"
(gdb)
```

The `-gdb-version` Command

Synopsis

```
 -gdb-version
```

Show version information for GDB. Used mostly in testing.

GDB command

The corresponding GDB command is '`show version`'.

Example

```
(gdb)
-gdb-version
~GNU gdb 5.2.1
~Copyright 2000 Free Software Foundation, Inc.
```

```
~GDB is free software, covered by the GNU General Public License, and
~you are welcome to change it and/or distribute copies of it under
~ certain conditions.
~Type "show copying" to see the conditions.
~There is absolutely no warranty for GDB.  Type "show warranty" for
~ details.
~This GDB was configured as
 "--host=sparc-sun-solaris2.5.1 --target=ppc-eabi".
^done
(gdb)
```

21.9 GDB/MI Stack Manipulation Commands

The `-stack-info-frame` command

Synopsis

```
 -stack-info-frame
```

Get info on the current frame.

GDB command

The corresponding GDB command is '`info frame`' or '`frame`' (without arguments).

Example

N.A.

The `-stack-info-depth` Command

Synopsis

```
 -stack-info-depth [ max-depth ]
```

Return the depth of the stack. If the integer argument *max-depth* is specified, do not count beyond *max-depth* frames.

GDB command

There no equivalent GDB command.

Example

For a stack with frame levels 0 through 11:

```
(gdb)
-stack-info-depth
^done,depth="12"
(gdb)
-stack-info-depth 4
^done,depth="4"
(gdb)
-stack-info-depth 12
^done,depth="12"
(gdb)
-stack-info-depth 11
^done,depth="11"
(gdb)
-stack-info-depth 13
^done,depth="12"
(gdb)
```

The `-stack-list-arguments` Command

Synopsis

```
 -stack-list-arguments show-values
    [ low-frame high-frame ]
```

Display a list of the arguments for the frames between *low-frame* and *high-frame* (inclusive). If *low-frame* and *high-frame* are not provided, list the arguments for the whole call stack.

The *show-values* argument must have a value of 0 or 1. A value of 0 means that only the names of the arguments are listed, a value of 1 means that both names and values of the arguments are printed.

GDB command

GDB does not have an equivalent command. `gdbtk` has a '`gdb_get_args`' command which partially overlaps with the functionality of '`-stack-list-arguments`'.

Example

```
(gdb)
-stack-list-frames
^done,
stack=[
frame={level="0 ",addr="0x00010734",func="callee4",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="8"},
frame={level="1 ",addr="0x0001076c",func="callee3",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="17"},
frame={level="2 ",addr="0x0001078c",func="callee2",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="22"},
frame={level="3 ",addr="0x000107b4",func="callee1",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="27"},
frame={level="4 ",addr="0x000107e0",func="main",
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="32"}]
(gdb)
-stack-list-arguments 0
^done,
stack-args=[
frame={level="0",args=[]},
frame={level="1",args=[name="strarg"]},
frame={level="2",args=[name="intarg",name="strarg"]},
frame={level="3",args=[name="intarg",name="strarg",name="fltarg"]},
frame={level="4",args=[]}]
(gdb)
-stack-list-arguments 1
^done,
stack-args=[
frame={level="0",args=[]},
frame={level="1",
 args=[{name="strarg",value="0x11940 \"A string argument.\""}]},
frame={level="2",args=[
{name="intarg",value="2"},
{name="strarg",value="0x11940 \"A string argument.\""}]},
{frame={level="3",args=[
{name="intarg",value="2"},
{name="strarg",value="0x11940 \"A string argument.\""},
{name="fltarg",value="3.5"}]},
frame={level="4",args=[]}]
(gdb)
-stack-list-arguments 0 2 2
^done,stack-args=[frame={level="2",args=[name="intarg",name="strarg"]}]
(gdb)
-stack-list-arguments 1 2 2
^done,stack-args=[frame={level="2",
args=[{name="intarg",value="2"},
{name="strarg",value="0x11940 \"A string argument.\""}]}]
(gdb)
```

The `-stack-list-frames` command

Synopsis

```
 -stack-list-frames [ low-frame high-frame ]
```

List the frames currently on the stack. For each frame it displays the following info:

'*level*' The frame number, 0 being the topmost frame, i.e. the innermost function.

'*addr*' The `$pc` value for that frame.

'*func*' Function name.

'*file*' File name of the source file where the function lives.

'*line*' Line number corresponding to the `$pc`.

If invoked without arguments, this command prints a backtrace for the whole stack. If given two integer arguments, it shows the frames whose levels are between the two arguments (inclusive). If the two arguments are equal, it shows the single frame at the corresponding level.

GDB command

The corresponding GDB commands are '`backtrace`' and '`where`'.

Example

Full stack backtrace:

```
(gdb)
-stack-list-frames
^done,stack=
[frame={level="0 ",addr="0x0001076c",func="foo",
  file="recursive2.c",line="11"},
frame={level="1 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="2 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="4 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="5 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="6 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="7 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="8 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="9 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="10",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="11",addr="0x00010738",func="main",
  file="recursive2.c",line="4"}]
(gdb)
```

Show frames between *low_frame* and *high_frame*:

```
(gdb)
-stack-list-frames 3 5
^done,stack=
[frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="4 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"},
frame={level="5 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"}]
(gdb)
```

Show a single frame:

```
(gdb)
-stack-list-frames 3 3
^done,stack=
[frame={level="3 ",addr="0x000107a4",func="foo",
  file="recursive2.c",line="14"}]
(gdb)
```

The `-stack-list-locals` command

Synopsis

```
 -stack-list-locals print-values
```

Display the local variable names for the current frame. With an argument of 0 prints only the names of the variables, with argument of 1 prints also their values.

GDB Command

'`info locals`' in GDB, '`gdb_get_locals`' in `gdbtk`.

Example

```
(gdb)
-stack-list-locals 0
^done,locals=[name="A",name="B",name="C"]
(gdb)
-stack-list-locals 1
^done,locals=[{name="A",value="1"},{name="B",value="2"},
  {name="C",value="3"}]
(gdb)
```

The `-stack-select-frame` Command

Synopsis

```
 -stack-select-frame framenum
```

Change the current frame. Select a different frame *framenum* on the stack.

GDB command

The corresponding GDB commands are '`frame`', '`up`', '`down`', '`select-frame`', '`up-silent`', and '`down-silent`'.

Example

```
(gdb)
-stack-select-frame 2
^done
(gdb)
```

21.10 GDB/MI Symbol query commands

The `-symbol-info-address` Command

Synopsis

```
 -symbol-info-address symbol
```

Describe where *symbol* is stored.

GDB command

The corresponding GDB command is '`info address`'.

Example

N.A.

The `-symbol-info-file` Command

Synopsis

```
 -symbol-info-file
```

Show the file for the symbol.

GDB command

There is no equivalent GDB command. `gdbtk` has '`gdb_find_file`'.

Example

N.A.

The `-symbol-info-function` Command

Synopsis

```
    -symbol-info-function
```

Show which function the symbol lives in.

GDB command

'`gdb_get_function`' in `gdbtk`.

Example

N.A.

The `-symbol-info-line` command

Synopsis

```
    -symbol-info-line
```

Show the core addresses of the code for a source line.

GDB command

The corresponding GDB comamnd is '`info line`'. `gdbtk` has the '`gdb_get_line`' and '`gdb_get_file`' commands.

Example

N.A.

The `-symbol-info-symbol` command

Synopsis

```
    -symbol-info-symbol addr
```

Describe what symbol is at location *addr*.

GDB command

The corresponding GDB command is '`info symbol`'.

Example

N.A.

The `-symbol-list-functions` command

Synopsis

```
     -symbol-list-functions
```

List the functions in the executable.

GDB command

'`info functions`' in GDB, '`gdb_listfunc`' and '`gdb_search`' in `gdbtk`.

Example

N.A.

The `-symbol-list-types` command

Synopsis

```
     -symbol-list-types
```

List all the type names.

GDB command

The corresponding commands are '`info types`' in GDB, '`gdb_search`' in `gdbtk`.

Example

N.A.

The `-symbol-list-variables` command

Synopsis

```
     -symbol-list-variables
```

List all the global and static variable names.

GDB command

'`info variables`' in GDB, '`gdb_search`' in `gdbtk`.

Example

N.A.

The `-symbol-locate` command

Synopsis

```
     -symbol-locate
```

GDB command

'`gdb_loc`' in `gdbtk`.

Example

N.A.

The `-symbol-type` command

Synopsis

```
     -symbol-type variable
```

Show type of *variable*.

GDB command

The corresponding GDB command is '`ptype`', `gdbtk` has '`gdb_obj_variable`'.

Example

N.A.

21.11 GDB/MI Target Manipulation Commands

The `-target-attach` command

Synopsis

```
     -target-attach pid | file
```

Attach to a process *pid* or a file *file* outside of GDB.

GDB command

The corresponding GDB command is '`attach`'.

Example

N.A.

The `-target-compare-sections` command

Synopsis

```
    -target-compare-sections [ section ]
```

Compare data of section *section* on target to the exec file. Without the argument, all sections are compared.

GDB command

The GDB equivalent is '`compare-sections`'.

Example

N.A.

The `-target-detach` command

Synopsis

```
    -target-detach
```

Disconnect from the remote target. There is no output.

GDB command

The corresponding GDB command is '`detach`'.

Example

```
(gdb)
-target-detach
^done
(gdb)
```

The `-target-download` command

Synopsis

```
-target-download
```

Loads the executable onto the remote target. It prints out an update message every half second, which includes the fields:

'`section`' The name of the section.

'`section-sent`'
: The size of what has been sent so far for that section.

'`section-size`'
: The size of the section.

'`total-sent`'
: The total size of what was sent so far (the current and the previous sections).

'`total-size`'
: The size of the overall executable to download.

Each message is sent as status record (see Section 21.1.2 [GDB/MI Output Syntax], page 270).

In addition, it prints the name and size of the sections, as they are downloaded. These messages include the following fields:

'`section`' The name of the section.

'`section-size`'
: The size of the section.

'`total-size`'
: The size of the overall executable to download.

At the end, a summary is printed.

GDB command

The corresponding GDB command is '`load`'.

Example

Note: each status message appears on a single line. Here the messages have been broken down so that they can fit onto a page.

```
(gdb)
-target-download
+download,{section=".text",section-size="6668",total-size="9880"}
+download,{section=".text",section-sent="512",section-size="6668",
total-sent="512",total-size="9880"}
+download,{section=".text",section-sent="1024",section-size="6668",
total-sent="1024",total-size="9880"}
+download,{section=".text",section-sent="1536",section-size="6668",
total-sent="1536",total-size="9880"}
+download,{section=".text",section-sent="2048",section-size="6668",
total-sent="2048",total-size="9880"}
+download,{section=".text",section-sent="2560",section-size="6668",
total-sent="2560",total-size="9880"}
+download,{section=".text",section-sent="3072",section-size="6668",
total-sent="3072",total-size="9880"}
+download,{section=".text",section-sent="3584",section-size="6668",
total-sent="3584",total-size="9880"}
+download,{section=".text",section-sent="4096",section-size="6668",
total-sent="4096",total-size="9880"}
+download,{section=".text",section-sent="4608",section-size="6668",
total-sent="4608",total-size="9880"}
+download,{section=".text",section-sent="5120",section-size="6668",
total-sent="5120",total-size="9880"}
+download,{section=".text",section-sent="5632",section-size="6668",
total-sent="5632",total-size="9880"}
+download,{section=".text",section-sent="6144",section-size="6668",
total-sent="6144",total-size="9880"}
+download,{section=".text",section-sent="6656",section-size="6668",
total-sent="6656",total-size="9880"}
+download,{section=".init",section-size="28",total-size="9880"}
+download,{section=".fini",section-size="28",total-size="9880"}
+download,{section=".data",section-size="3156",total-size="9880"}
+download,{section=".data",section-sent="512",section-size="3156",
total-sent="7236",total-size="9880"}
+download,{section=".data",section-sent="1024",section-size="3156",
total-sent="7748",total-size="9880"}
+download,{section=".data",section-sent="1536",section-size="3156",
total-sent="8260",total-size="9880"}
+download,{section=".data",section-sent="2048",section-size="3156",
total-sent="8772",total-size="9880"}
+download,{section=".data",section-sent="2560",section-size="3156",
total-sent="9284",total-size="9880"}
+download,{section=".data",section-sent="3072",section-size="3156",
total-sent="9796",total-size="9880"}
^done,address="0x10004",load-size="9880",transfer-rate="6586",
write-rate="429"
(gdb)
```

The `-target-exec-status` command

Synopsis

```
 -target-exec-status
```

Provide information on the state of the target (whether it is running or not, for instance).

GDB command

There is no equivalent GDB command.

Example

N.A.

The `-target-list-available-targets` command

Synopsis

```
    -target-list-available-targets
```

List the possible targets to connect to.

GDB command

The corresponding GDB command is '`help target`'.

Example

N.A.

The `-target-list-current-targets` command

Synopsis

```
    -target-list-current-targets
```

Describe the current target.

GDB command

The corresponding information is printed by '`info file`' (among other things).

Example

N.A.

The `-target-list-parameters` command

Synopsis

```
    -target-list-parameters
```

GDB command

No equivalent.

Example

N.A.

The `-target-select` command

Synopsis

```
    -target-select type parameters ...
```

Connect GDB to the remote target. This command takes two args:

‘*type*’ The type of target, for instance ‘`async`’, ‘`remote`’, etc.

‘*parameters*’
: Device names, host names and the like. See Section 13.2 [Commands for managing targets], page 109, for more details.

The output is a connection notification, followed by the address at which the target program is, in the following form:

```
^connected,addr="address",func="function name",
  args=[arg list]
```

GDB command

The corresponding GDB command is ‘`target`’.

Example

```
(gdb)
-target-select async /dev/ttya
^connected,addr="0xfe00a300",func="??",args=[]
(gdb)
```

21.12 GDB/MI thread commands

The `-thread-info` command

Synopsis

```
    -thread-info
```

GDB command

No equivalent.

Example

N.A.

The `-thread-list-all-threads` Command

Synopsis

```
-thread-list-all-threads
```

GDB command

The equivalent GDB command is '`info threads`'.

Example

N.A.

The `-thread-list-ids` command

Synopsis

```
-thread-list-ids
```

Produces a list of the currently known GDB thread ids. At the end of the list it also prints the total number of such threads.

GDB command

Part of '`info threads`' supplies the same information.

Example

No threads present, besides the main process:

```
(gdb)
-thread-list-ids
^done,thread-ids={},number-of-threads="0"
(gdb)
```

Several threads:

```
(gdb)
-thread-list-ids
^done,thread-ids={thread-id="3",thread-id="2",thread-id="1"},
number-of-threads="3"
(gdb)
```

The `-thread-select` command

Synopsis

```
    -thread-select threadnum
```

Make *threadnum* the current thread. It prints the number of the new current thread, and the topmost frame for that thread.

GDB command

The corresponding GDB command is '`thread`'.

Example

```
(gdb)
-exec-next
^running
(gdb)
~"0x7f7f0aec"
*stopped,reason="end-stepping-range",thread-id="2",frame=addr="0x00002ca4",func
="printme",args=[name="ip"],file="multithread.c",line="9"
(gdb)
-thread-list-ids
^done,thread-ids=thread-id="2",thread-id="1",number-of-threads="2"
(gdb)
-thread-select 1
^done,new-thread-id="1",frame=level="0 ",addr="0x7ad47d70",func="_lwp_create","
+0x10",args=[],from="/usr/lib/libpthread.1"
(gdb)
```

21.13 GDB/MI tracepoint commands

The tracepoint commands are not yet implemented.

21.14 GDB/MI variable objects

Motivation for variable objects in GDB/MI

For the implementation of a variable debugger window (locals, watched expressions, etc.), we are proposing the adaptation of the existing code used by `Insight`.

The two main reasons for that are:

1. It has been proven in practice (it is already on its second generation).
2. It will shorten development time (needless to say how important it is now).

The original interface was designed to be used by Tcl code, so it was slightly changed so it could be used through GDB/MI. This section describes the GDB/MI operations that will be available and gives some hints about their use.

Note: In addition to the set of operations described here, we expect the GUI implementation of a variable window to require, at least, the following operations:

- `-gdb-show output-radix`
- `-stack-list-arguments`
- `-stack-list-locals`
- `-stack-select-frame`

Introduction to variable objects in GDB/MI

The basic idea behind variable objects is the creation of a named object to represent a variable, an expression, a memory location or even a CPU register. For each object created, a set of operations is available for examining or changing its properties.

Furthermore, complex data types, such as C structures, are represented in a tree format. For instance, the `struct` type variable is the root and the children will represent the struct members. If a child is itself of a complex type, it will also have children of its own. Appropriate language differences are handled for C, C++ and Java.

When returning the actual values of the objects, this facility allows for the individual selection of the display format used in the result creation. It can be chosen among: binary, decimal, hexadecimal, octal and natural. Natural refers to a default format automatically chosen based on the variable type (like decimal for an `int`, hex for pointers, etc.).

The following is the complete set of GDB/MI operations defined to access this functionality:

Operation	**Description**
`-var-create`	create a variable object
`-var-delete`	delete the variable object and its children
`-var-set-format`	set the display format of this variable
`-var-show-format`	show the display format of this variable
`-var-info-num-children`	tells how many children this object has
`-var-list-children`	return a list of the object children
`-var-info-type`	show the type of this variable object
`-var-info-expression`	print what this variable object represents
`-var-show-attributes`	is this variable editable? does it exist here?
`-var-evaluate-expression`	get the value of this variable
`-var-assign`	set the value of this variable
`-var-update`	update the variable and its children

In the next subsection we describe each operation in detail and suggest how it can be used.

Description and use of operations on variable objects

The `-var-create` command

Synopsis

```
    -var-create {name | "-"}
       {frame-addr | "*"} expression
```

This operation creates a variable object, which allows the monitoring of a variable, the result of an expression, a memory cell or a CPU register.

The *name* parameter is the string by which the object can be referenced. It must be unique. If '`-`' is specified, the varobj system will generate a string "varNNNNNN" automatically. It will be unique provided that one does not specify *name* on that format. The command fails if a duplicate name is found.

The frame under which the expression should be evaluated can be specified by *frame-addr*. A '`*`' indicates that the current frame should be used.

expression is any expression valid on the current language set (must not begin with a '`*`'), or one of the following:

- '`*addr`', where *addr* is the address of a memory cell
- '`*addr-addr`' — a memory address range (TBD)
- '`$regname`' — a CPU register name

Result

This operation returns the name, number of children and the type of the object created. Type is returned as a string as the ones generated by the GDB CLI:

```
    name="name",numchild="N",type="type"
```

The `-var-delete` Command

Synopsis

```
    -var-delete name
```

Deletes a previously created variable object and all of its children.

Returns an error if the object *name* is not found.

The `-var-set-format` command

Synopsis

```
    -var-set-format name format-spec
```

Sets the output format for the value of the object *name* to be *format-spec*.

The syntax for the *format-spec* is as follows:

```
 format-spec ↦
 {binary | decimal | hexadecimal | octal | natural}
```

The `-var-show-format` command

Synopsis

```
     -var-show-format name
```

Returns the format used to display the value of the object *name*.

```
     format ↦
     format-spec
```

The `-var-info-num-children` command

Synopsis

```
     -var-info-num-children name
```

Returns the number of children of a variable object *name*:

```
     numchild=n
```

The `-var-list-children` command

Synopsis

```
     -var-list-children name
```

Returns a list of the children of the specified variable object:

```
     numchild=n,children={{name=name,
     numchild=n,type=type},(repeats N times)}
```

The `-var-info-type` command

Synopsis

```
     -var-info-type name
```

Returns the type of the specified variable *name*. The type is returned as a string in the same format as it is output by the GDB CLI:

```
     type=typename
```

The `-var-info-expression` command

Synopsis

```
-var-info-expression name
```

Returns what is represented by the variable object *name*:

```
lang=lang-spec,exp=expression
```

where *lang-spec* is `{"C" | "C++" | "Java"}`.

The `-var-show-attributes` command

Synopsis

```
-var-show-attributes name
```

List attributes of the specified variable object *name*:

```
status=attr [ ( ,attr )* ]
```

where *attr* is `{ { editable | noneditable } | TBD }`.

The `-var-evaluate-expression` command

Synopsis

```
-var-evaluate-expression name
```

Evaluates the expression that is represented by the specified variable object and returns its value as a string in the current format specified for the object:

```
value=value
```

The `-var-assign` Command

Synopsis

```
-var-assign name expression
```

Assigns the value of *expression* to the variable object specified by *name*. The object must be '`editable`'.

The `-var-update` Command

Synopsis

```
-var-update {name | "*"}
```

Update the value of the variable object *name* by evaluating its expression after fetching all the new values from memory or registers. A '`*`' causes all existing variable objects to be updated.

22 Reporting Bugs in GDB

Your bug reports play an essential role in making GDB reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of GDB work better. Bug reports are your contribution to the maintenance of GDB.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

22.1 Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the debugger gets a fatal signal, for any input whatever, that is a GDB bug. Reliable debuggers never crash.
- If GDB produces an error message for valid input, that is a bug. (Note that if you're cross debugging, the problem may also be somewhere in the connection to the target.)
- If GDB does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of debugging tools, your suggestions for improvement of GDB are welcome in any case.

22.2 How to report bugs

If you obtained GDB (Hewlett-Packard Wildebeest (based on GDB 5.0-hpwdb-20000516)) as part of your HP ANSI C, HP ANSI C++, or HP Fortran compiler kit, report problems to your HP Support Representative.

If you obtained GDB (Hewlett-Packard Wildebeest (based on GDB 5.0-hpwdb-20000516)) from the Hewlett-Packard Web site, report problems to your HP Support Representative. Support is covered under the support contract for your HP compiler.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the debugger into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug. It may be that the bug has been reported previously, but neither you nor we can know that unless your bug report is complete and self-contained.

Sometimes people give a few sketchy facts and ask, “Does this ring a bell?” Those bug reports are useless, and we urge everyone to *refuse to respond to them* except to chide the sender to report bugs properly.

To enable us to fix the bug, you should include all these things:

- The version of GDB. GDB announces it if you start with no arguments; you can also print it at any time using `show version`.

  Without this, we will not know whether there is any point in looking for the bug in the current version of GDB.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile the program you are debugging—e.g. “HP92453-01 A.10.32.03 HP C Compiler”. Use the `what` command with the pathname of the compile command (‘`what /opt/ansic/bin/cc`’, for example) to obtain this information.
- The command arguments you gave the compiler to compile your example and observe the bug. For example, did you use ‘`-O`’? To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

  If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.
- A complete input script, and all necessary source files, that will reproduce the bug.
- A description of what behavior you observe that you believe is incorrect. For example, “It gets a fatal signal.”

  Of course, if the bug is that GDB gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of GDB is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

  However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

  A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GDB it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

  And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.
- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

Appendix A Installing GDB

If you obtain GDB (WDB) as part of the HP ANSI C, HP ANSI C++ Developer's Kit for HP-UX Release 11.x, or HP Fortran, you do not have to take any special action to build or install GDB.

If you obtain GDB (WDB) from an HP web site, you may download either an `swinstall` package or a source tree, or both.

Most customers will want to install the GDB binary that is part of the `swinstall` package. To do so, use a command of the form

```
/usr/sbin/swinstall -s package-name WDB
```

Alternatively, it is possible to build GDB from the source distribution. If you want to modify the debugger sources to tailor GDB to your needs, you may wish to do this. The source distribution consists of a `tar` file containing the source tree rooted at '`gdb-4.17/...`'. The instructions that follow describe how to build a '`gdb`' executable from this source tree. HP believes that these instructions apply to the WDB source tree that it distributes. However, HP does not explicitly support building a '`gdb`' for any non-HP platform from the WDB source tree. It may work, but HP has not tested it for any platforms other than those described in the WDB Release Notes.

You can find additional information specific to Hewlett-Packard in the '`README.HP.WDB`' file at the root of the source tree.

GDB comes with a `configure` script that automates the process of preparing GDB for installation; you can then use `make` to build the `gdb` program.[1]

The GDB distribution includes all the source code you need for GDB in a single directory, whose name is usually composed by appending the version number to '`gdb`'.

For example, the GDB version gdb-199991101 distribution is in the '`gdb-gdb-199991101`' directory. That directory contains:

`gdb-gdb-199991101/configure` (and supporting files)
: script for configuring GDB and all its supporting libraries

`gdb-gdb-199991101/gdb`
: the source specific to GDB itself

`gdb-gdb-199991101/bfd`
: source for the Binary File Descriptor library

`gdb-gdb-199991101/include`
: GNU include files

`gdb-gdb-199991101/libiberty`
: source for the '`-liberty`' free software library

`gdb-gdb-199991101/opcodes`
: source for the library of opcode tables and disassemblers

`gdb-gdb-199991101/readline`
: source for the GNU command-line interface

[1] If you have a more recent version of GDB than gdb-199991101, look at the '`README`' file in the sources; we may have improved the installation procedures since publishing this manual.

`gdb-gdb-199991101/glob`
: source for the GNU filename pattern-matching subroutine

`gdb-gdb-199991101/mmalloc`
: source for the GNU memory-mapped malloc package

The simplest way to configure and build GDB is to run `configure` from the '`gdb-`*`version-number`*' source directory, which in this example is the '`gdb-gdb-199991101`' directory.

First switch to the '`gdb-`*`version-number`*' source directory if you are not already in it; then run `configure`. Pass the identifier for the platform on which GDB will run as an argument.

For example:

```
cd gdb-gdb-199991101
./configure host
make
```

where *host* is an identifier such as '`sun4`' or '`decstation`', that identifies the platform where GDB will run. (You can often leave off *host*; `configure` tries to guess the correct value by examining your system.)

Running '`configure` *`host`*' and then running `make` builds the '`bfd`', '`readline`', '`mmalloc`', and '`libiberty`' libraries, then `gdb` itself. The configured source files, and the binaries, are left in the corresponding source directories.

`configure` is a Bourne-shell (`/bin/sh`) script; if your system does not recognize this automatically when you run a different shell, you may need to run `sh` on it explicitly:

```
sh configure host
```

If you run `configure` from a directory that contains source directories for multiple libraries or programs, such as the '`gdb-gdb-199991101`' source directory for version gdb-199991101, `configure` creates configuration files for every directory level underneath (unless you tell it not to, with the '`--norecursion`' option).

You can run the `configure` script from any of the subordinate directories in the GDB distribution if you only want to configure that subdirectory, but be sure to specify a path to it.

For example, with version gdb-199991101, type the following to configure only the `bfd` subdirectory:

```
cd gdb-gdb-199991101/bfd
../configure host
```

You can install `(gdb)` anywhere; it has no hardwired paths. However, you should make sure that the shell on your path (named by the '`SHELL`' environment variable) is publicly readable. Remember that GDB uses the shell to start your program—some systems refuse to let GDB debug child processes whose programs are not readable.

A.1 Compiling GDB in another directory

If you want to run GDB versions for several host or target machines, you need a different `gdb` compiled for each combination of host and target. `configure` is designed to make this easy by allowing you to generate each configuration in a separate subdirectory, rather than in the source directory. If your `make` program handles the '`VPATH`' feature (GNU `make` does), running `make` in each of these directories builds the `gdb` program specified there.

To build `gdb` in a separate directory, run `configure` with the '`--srcdir`' option to specify where to find the source. (You also need to specify a path to find `configure` itself from your working directory. If the path to `configure` would be the same as the argument to '`--srcdir`', you can leave out the '`--srcdir`' option; it is assumed.)

For example, with version gdb-199991101, you can build GDB in a separate directory for a Sun 4 like this:

```
cd gdb-gdb-199991101
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-gdb-199991101/configure sun4
make
```

When `configure` builds a configuration using a remote source directory, it creates a tree for the binaries with the same structure (and using the same names) as the tree under the source directory. In the example, you'd find the Sun 4 library '`libiberty.a`' in the directory '`gdb-sun4/libiberty`', and GDB itself in '`gdb-sun4/gdb`'.

One popular reason to build several GDB configurations in separate directories is to configure GDB for cross-compiling (where GDB runs on one machine—the *host*—while debugging programs that run on another machine—the *target*). You specify a cross-debugging target by giving the '`--target=`*`target`*' option to `configure`.

When you run `make` to build a program or library, you must run it in a configured directory—whatever directory you were in when you called `configure` (or one of its subdirectories).

The `Makefile` that `configure` generates in each source directory also runs recursively. If you type `make` in a source directory such as '`gdb-gdb-199991101`' (or in a separate configured directory configured with '`--srcdir=`*`dirname`*`/gdb-gdb-199991101`'), you will build all the required libraries, and then build GDB.

When you have multiple hosts or targets configured in separate directories, you can run `make` on them in parallel (for example, if they are NFS-mounted on each of the hosts); they will not interfere with each other.

A.2 Specifying names for hosts and targets

The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, but some short predefined aliases are also supported. The full naming scheme encodes three pieces of information in the following pattern:

`architecture-vendor-os`

For example, you can use the alias `sun4` as a *host* argument, or as the value for *target* in a `--target=`*`target`* option. The equivalent full name is '`sparc-sun-sunos4`'.

The `configure` script accompanying GDB does not provide any query facility to list all supported host and target names or aliases. `configure` calls the Bourne shell script `config.sub` to map abbreviations to full names; you can read the script, if you wish, or you can use it to test your guesses on abbreviations—for example:

```
% sh config.sub i386-linux
i386-pc-linux-gnu
% sh config.sub alpha-linux
alpha-unknown-linux-gnu
% sh config.sub hp9k700
hppa1.1-hp-hpux
% sh config.sub sun4
sparc-sun-sunos4.1.1
% sh config.sub sun3
m68k-sun-sunos4.1.1
% sh config.sub i986v
Invalid configuration 'i986v': machine 'i986v' not recognized
```

`config.sub` is also distributed in the GDB source directory ('`gdb-gdb-199991101`', for version gdb-199991101).

A.3 `configure` options

Here is a summary of the `configure` options and arguments that are most often useful for building GDB. `configure` also has several other options not listed here. See Info file '`configure.info`', node '`What Configure Does`', for a full explanation of `configure`.

```
configure [--help]
          [--prefix=dir]
          [--exec-prefix=dir]
          [--srcdir=dirname]
          [--norecursion] [--rm]
          [--target=target]
          host
```

You may introduce options with a single '`-`' rather than '`--`' if you prefer; but you may abbreviate option names if you use '`--`'.

`--help` Display a quick summary of how to invoke `configure`.

`--prefix=`*`dir`*
: Configure the source to install programs and files under directory '*`dir`*'.

`--exec-prefix=`*`dir`*
: Configure the source to install programs under directory '*`dir`*'.

`--srcdir=`*`dirname`*
: **Warning: using this option requires GNU `make`, or another `make` that implements the `VPATH` feature.**
Use this option to make configurations in directories separate from the GDB source directories. Among other things, you can use this to build (or maintain) several configurations simultaneously, in separate directories. `configure` writes configuration specific files in the current directory, but arranges for them to use the source in the directory *dirname*. `configure` creates directories under the working directory in parallel to the source directories below *dirname*.

`--norecursion`
: Configure only the directory level where `configure` is executed; do not propagate configuration to subdirectories.

`--target=`*`target`*
: Configure GDB for cross-debugging programs running on the specified *target*. Without this option, GDB is configured to debug programs that run on the same machine (*host*) as GDB itself.

  There is no convenient way to generate a list of all available targets.

`host` `...`
: Configure GDB to run on the specified *host*.

  There is no convenient way to generate a list of all available hosts.

There are many other options available as well, but they are generally needed for special purposes only.

\enddocument

Index

(Index is nonexistent)

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt,
cmb10 at 10.95pt, and
cmsl10 at 10.95pt
are used for emphasis.