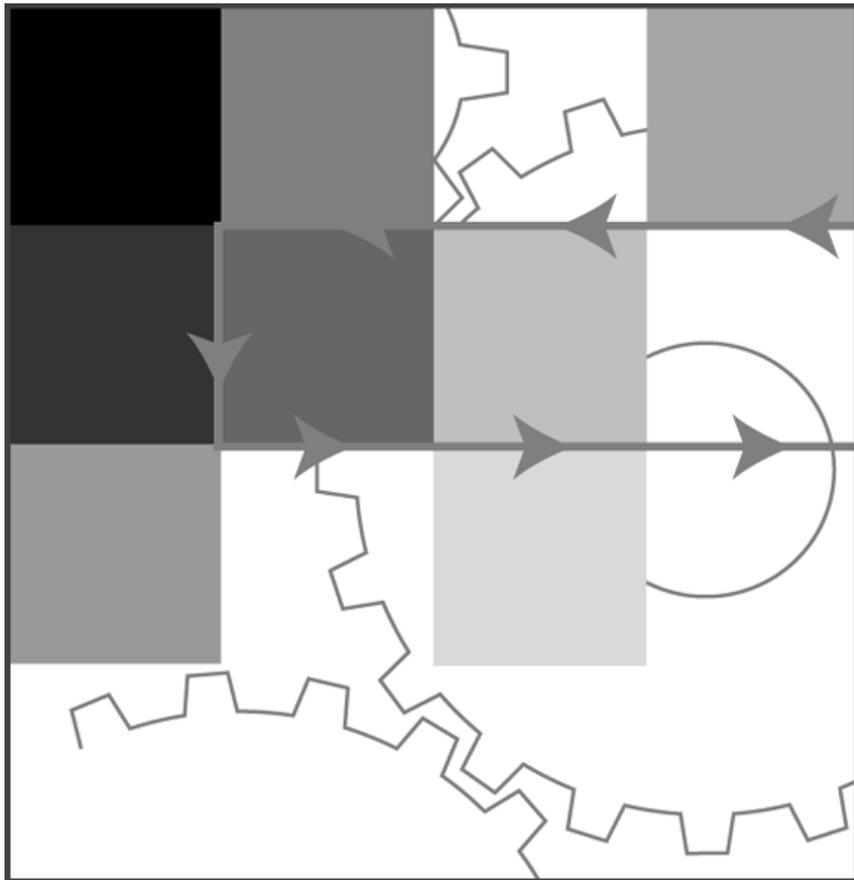


GNUPro® Toolkit

GNUPro Debugging Tools

- ***Debugging with GDB***
- ***Insight, the GNUPro Debugger GUI Interface***



GNUPro 2001

Copyright © 1991-2001 Red Hat[®], Inc. All rights reserved.

Red Hat[®], GNUPro[®], the Red Hat Shadow Man logo[®], Source-Navigator[™], Insight[™], Cygwin[™], and eCos[™], and Red Hat Embedded DevKit[™] are all trademarks or registered trademarks of Red Hat, Inc. ARM[®], Thumb[®], and ARM Powered[®] are registered trademarks of ARM Limited. SA[™], SA-110[™], SA-1100[™], SA-1110[™], SA-1500[™], SA-1510[™] are trademarks of ARM Limited. All other brands or product names are the property of their respective owners. “ARM” is used to represent any or all of ARM Holdings plc (LSE; ARM: NASDAQ; ARMHY), its operating company, ARM Limited, and the regional subsidiaries ARM INC., ARM KK, and ARM Korea Ltd.

AT&T[®] is a registered trademark of AT&T, Inc.

Hitachi[®], SuperH[®], and H8[®] are registered trademarks of Hitachi, Ltd.

IBM[®], PowerPC[®], and RS/6000[®] are registered trademarks of IBM Corporation.

Intel[®], Pentium[®], Pentium II[®], and StrongARM[®] are registered trademarks of Intel Corporation.

Linux[®] is a registered trademark of Linus Torvalds.

Matsushita[®], Panasonic[®], PanaX[®], and PanaXSeries[®] are registered trademarks of Matsushita, Inc.

Microsoft[®] Windows[®] CE, Microsoft[®] Windows NT[®], Microsoft[®] Windows[®] 98, and Win32[®] are registered trademarks of Microsoft Corporation.

MIPS[®] is a registered trademark and MIPS I[™], MIPS II[™], MIPS III[™], MIPS IV[™], and MIPS16[™] are all trademarks or registered trademarks of MIPS Technologies, Inc.

Mitsubishi[®] is a registered trademark of Mitsubishi Electric Corporation.

Motorola[®] is a registered trademark of Motorola, Inc.

Sun[®], SPARC[®], SunOS[™], Solaris[™], and Java[™], are trademarks or registered trademarks of Sun Microsystems, Inc..

UNIX[®] is a registered trademark of The Open Group.

NEC[®], VR5000[™], VRC5074[™], VR5400[™], VR5432[™], VR5464[™], VRC5475[™], VRC5476[™], VRC5477[™], VRC5484[™] are trademarks or registered trademarks of NEC Corporation.

All other brand and product names, services names, trademarks and copyrights are the property of their respective owners.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation. For licenses and use information, see “General Licenses and Terms for Using GNUPro Toolkit” in the GNUPro Toolkit *Getting Started Guide*.

How to Contact Red Hat

Use the following means to contact Red Hat.

Red Hat Corporate Headquarters

2600 Meridian Parkway

Durham, NC 27713 USA

Telephone (toll free): +1 888 REDHAT 1

Telephone (main line): +1 919 547 0012

Telephone (FAX line): +1 919 547 0024

Website: <http://www.redhat.com/>

Contents

Overview of GNUPro Debugger Tools	1
---	---

Debugging with GDB

Summary of GDB, the GNU Debugger	5
GDB as Free Software	6
Requirements of GDB.....	7
Contributors to GDB.....	7
Overall Structure of GDB	10
Configuring GDB.....	10
Symbol Handling for GDB	11
Symbol Reading.....	11
Partial Symbol Tables	12
Types.....	14
Object File Formats for GDB.....	14
Debugging File Formats	15
Adding a New Symbol Reader to GDB	16
Installing GDB	17
Locating Files for Installing GDB.....	18
Compiling GDB in Another Directory.....	19

Specifying Names for Hosts and Targets	20
configure Options with GDB	20
Essentials of GDB	23
Invoking GDB	23
Choosing Files for GDB to Debug	24
Choosing Modes	26
Quitting GDB	27
Shell Commands for GDB	27
GDB Commands	29
Command Syntax	29
Command Completion	30
Getting Help	32
Running Programs under GDB	35
Compiling for Debugging	36
Starting a Program	36
Your Program's Arguments	37
Your Program's Environment	38
Your Program's Working Directory	39
Your Program's Input and Output	39
Debugging a Running Process	40
Killing the Child Process	41
Additional Process Information	41
Debugging Programs with Multiple Threads	42
Debugging Programs with Multiple Processes	44
Stopping and Continuing	45
Breakpoints, Watchpoints, and Exceptions	46
Setting Breakpoints	47
Setting Watchpoints	50
Setting Catchpoints	51
Deleting Breakpoints	52
Disabling Breakpoints	53
Break Conditions	54
Breakpoint Command Lists	56
Breakpoint Menus	57
Continuing and Stepping	58
Signals	60
Stopping and Starting Multiple Thread Programs	62
Examining the Stack	65
Stack Frames	66
Backtraces	67
Selecting a Frame	67
Information about a Frame	69

Examining Source Files	71
Printing Source Lines.....	71
Searching Source Files.....	73
Specifying Source Directories.....	74
Source and Machine Code	75
Examining Data	77
Expressions	78
Program Variables.....	78
Artificial Arrays	80
Output Formats.....	81
Examining Memory	82
Automatic Display	83
Print Settings	85
Value History	89
Convenience Variables.....	90
Registers	91
Floating Point Hardware	93
Using GDB with Different Languages	95
Switching between Source Languages.....	96
List of Filename Extensions and Languages.....	96
Setting GDB's Working Language.....	97
Having GDB Infer the Source Language.....	97
Displaying the Language.....	97
Type and Range Checking	98
An Overview of Type Checking	99
An Overview of Range Checking	100
Supported languages	101
C and C++	101
C and C++ Operators	101
C and C++ Constants	103
C++ Expressions.....	104
C and C++ Defaults	105
C and C++ Type and Range Checks.....	105
GDB and C.....	106
GDB Features for C++.....	106
Modula-2.....	107
Modula-2 Operators.....	107
Modula-2 Built-in Functions and Procedures.....	109
Modula-2 Constants	110
Modula-2 Defaults	111
Deviations from Standard Modula-2	111
Modula-2 Type and Range Checks.....	112

Modula-2 Scope Operator (.) and GDB Scope Operator (::)	112
GDB and Modula-2	112
Examining the Symbol Table	115
Altering Execution	119
Assignment to Variables	119
Continuing at a Different Address	120
Giving a Program a Signal	121
Returning from a Function	122
Calling Program Functions	122
Patching Programs	122
GDB Files	125
Commands to Specify Files	125
Errors Reading Symbol Files	129
Specifying a Debugging Target	131
Active Targets	131
Commands for Managing Targets	132
Choosing Target Byte Order	134
Remote Debugging	135
The GDB Remote Serial Protocol	135
What the Stub Can Do	137
What You Must Do for the Stub	137
Putting It All Together	139
Communication Protocol	140
Using the <code>gdbserver</code> Program	141
Using the <code>gdbserve.nlm</code> Program	143
GDB with a Remote i960 (Nindy)	144
Startup with Nindy	144
Nindy Reset Command	144
Options for Nindy	144
The UDI Protocol for AMD29K	145
The EBMON Protocol for AMD29K	145
GDB with a Tandem ST2000	148
GDB and VxWorks	148
GDB and SPARClet	150
Setting <code>file</code> to Debug	150
Connecting to SPARClet	151
GDB and Hitachi Microprocessors	152
Connecting to Hitachi Boards	152
Using the E7000 In-circuit Emulator	152
GDB and Remote MIPS Boards	153
Controlling GDB	157
Prompt	158

Command Editing	158
Command History	158
Screen Size	160
Numbers	160
Optional Warnings and Messages	161
Canned Sequences of Commands	163
User-defined Commands	163
User-defined Command Hooks	165
Command Files	165
Commands for Controlled Output	166

Insight, the GNUPro Debugger GUI

Insight, GDB's Alternative Interface	171
Using the Source Window	172
Using the Mouse in the Source Window	176
Source Window Menus and Display Features	179
Below the horizontal scroll bar of the Source Window	179
Using the Stack Window	182
Using the Registers Window	183
Using the Memory Window	184
Using the Watch Expressions Window	186
Using the Local Variables Window	188
Using the Breakpoints Window	191
Using the Console Window	194
Using the Function Browser Window	195
Using the Processes Window for Threads	197
Using the Help Window	198
Examples of Debugging with Insight	199
Selecting and Examining a Source File	200
Setting Breakpoints and Viewing Local Variables	202
Setting Breakpoints on Multiple Threads	206

Appendixes

Using GDB under GNU Emacs	211
Emacs Considerations with GDB	211
Keystroke Sequences for GDB with Emacs	212
Reporting Bugs in GDB	215
Have You Found a Bug?	215

How to Report Bugs.....	216
Command Line Editing	219
Readline Interaction	220
Readline Bare Essentials	220
Readline Movement Commands	221
Readline Killing Commands	221
Readline Arguments.....	222
Searching for Commands in the History	222
Readline <code>init</code> File	223
Readline <code>init</code> Syntax.....	223
Variable Settings for Readline	224
Key Bindings for Readline	225
Conditional <code>init</code> Constructs.....	227
Sample <code>init</code> File.....	228
Bindable Readline Commands.....	230
Commands for Moving around in Readline	230
Commands for Manipulating History with Readline.....	231
Commands for Changing Text in Readline	232
Killing and Yanking.....	233
Specifying Numeric Arguments	234
Letting Readline Type for You.....	235
Keyboard Macros.....	235
Some Miscellaneous Readline Commands.....	235
Readline in <code>vi</code> Mode	236
Using History Interactively	237
Event Designators	238
Word Designators	238
Modifiers	239
Formatting Documentation	241
Index	243

Overview of GNUPro Debugger Tools

The following documentation details “Debugging with GDB” in *GNUPro Debugger Tools*.

- “Summary of GDB, the GNU Debugger” on page 5
- “Installing GDB” on page 17
- “Essentials of GDB” on page 23
- “GDB Commands” on page 29
- “Running Programs under GDB” on page 35
- “Stopping and Continuing” on page 45
- “Examining the Stack” on page 65
- “Examining Source Files” on page 71
- “Examining Data” on page 77
- “Using GDB with Different Languages” on page 95
- “Examining the Symbol Table” on page 115
- “Altering Execution” on page 119
- “GDB Files” on page 125

- “Specifying a Debugging Target” on page 131
- “Controlling GDB” on page 157
- “Canned Sequences of Commands” on page 163

The following documentation details “Insight, the GNUPro Debugger GUI” in *GNUPro Debugger Tools*.

- “Insight, GDB’s Alternative Interface” on page 171
- “Examples of Debugging with Insight” on page 199

The following documentation details miscellaneous features of *GNUPro Debugger Tools*.

- “Using GDB under GNU Emacs” on page 211
- “Reporting Bugs in GDB” on page 215
- “Command Line Editing” on page 219
- “Using History Interactively” on page 237
- “Formatting Documentation” on page 241

Debugging with GDB

Copyright © 1991-2000 Free Software Foundation

All rights reserved.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation.

For licenses and use information, see *Getting Started Guide*.

1

Summary of GDB, the GNU Debugger

The purpose of a debugger such as the GNU debugger, 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 stopped. GDB can do four main kinds of things to help you catch “bugs.”

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another problem affecting your program.

The following documentation provides fundamental details about the GNU debugger, GDB.

- “Overall Structure of GDB” on page 10
- “Requirements of GDB” on page 7
- “Configuring GDB” on page 10
- “Symbol Handling for GDB” on page 11
- “Object File Formats for GDB” on page 14

The following documentation provides more details about the GNU debugger, GDB.

- “Installing GDB” on page 17 (only for developers who download source code)
- “Essentials of GDB” on page 23

- “GDB Commands” on page 29
- “Running Programs under GDB” on page 35
- “Stopping and Continuing” on page 45
- “Examining the Stack” on page 65
- “Examining Source Files” on page 71
- “Examining Data” on page 77
- “Using GDB with Different Languages” on page 95
- “Examining the Symbol Table” on page 115
- “Altering Execution” on page 119
- “GDB Files” on page 125
- “Specifying a Debugging Target” on page 131
- “Controlling GDB” on page 157
- “Canned Sequences of Commands” on page 163

See “Insight, the GNUPro Debugger GUI” on page 169 for documentation for the graphical user interface for GDB.

The following documentation details some miscellaneous features with *GNUPro Debugging Tools*.

- “Using GDB under GNU Emacs” on page 211
- “Reporting Bugs in GDB” on page 215
- “Command Line Editing” on page 219
- “Using History Interactively” on page 237
- “Formatting Documentation” on page 241

GDB as 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. To see the GNU General Public License, see “General Licenses and Terms for Using GNUPro Toolkit” on page 109 in *Getting Started Guide*.

Requirements of GDB

Before using GDB, you should understand the formal requirements and other expectations for GDB. Although some of these may seem obvious, there have been proposals for GDB that have run counter to these requirements.

First of all, GDB is a debugger. It's not designed to be a front panel for embedded systems. It's not a text editor. It's not a shell. It's not a programming environment.

GDB is an interactive tool. Although a batch mode is available, GDB's primary role is to interact with a human programmer.

GDB should be responsive to the user. A programmer hot on the trail of a nasty bug, and operating under a looming deadline, is going to be very impatient of everything, including the response time to debugger commands.

GDB should be relatively permissive, such as for expressions. While the compiler should be picky (or have the option to be made picky), since source code lives for a long time usually, the programmer doing debugging shouldn't be spending time figuring out to mollify the debugger.

GDB will be called upon to deal with really large programs. Executable sizes of 50 to 100 megabytes occur regularly, and there are reports of programs approaching 1 gigabyte in size.

GDB should be able to run everywhere. No other debugger is available for even half as many configurations as GDB supports.

Contributors to GDB

Richard Stallman was the original author of GDB, among other GNU programs. Many others have contributed to its development, and this section attempts to credit the major contributors. One of the virtues of free software is that everyone is free to contribute to it; with regret, we cannot acknowledge everyone here. The 'ChangeLog' file in the GDB distribution approximates a blow-by-blow account. Changes much prior to version 2.0 are lost in the mists of time.

IMPORTANT! 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 long labor as thankless, we particularly thank those who shepherded GDB through major releases: Jim Blandy (release 4.18), Jason Molenda (release 4.17), Stan Shebs (releases 4.1.4, 4.1.5, 4.1.6 and 4.1.7), Fred Fish (releases 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). As major maintainer of GDB for some period, each contributed significantly to the structure, stability, and capabilities of the entire debugger.

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.

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 CXUX 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.

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 and Wind River Systems contributed remote debugging modules for their products.

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 Debugging with GDB.

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 Hitachi microprocessors.

NEC sponsored the support for the V850, V_R4_{xxx}, and V_R5_{xxx} 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 SPARC_{lite} 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 11.0 (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 provided HP-specific information in this documentation.

Cygnus, a Red Hat company, sponsored GDB maintenance and much of its development since 1991. Cygnus engineers who have worked on GDB include Mark Alexander, Jim Blandy, Per Bothner, Michael Chastain, 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, Jim Ingham, Jeff Law, Jim Lemke, Tom Lord, Bob Manson, Michael Meissner, Jason Merrill, Catherine Moore, Drew Moseley, Ken Raeburn, Gavin Romig-Koch, Rob Savoye,

Keith Seitz, Jamie Smith, Michael Snyder, Stan Shebs, Mike Stump, Ian Taylor, Angela Thomas, Michael Tiemann, Tom Tromey, Ron Unrau, Jim Wilson, and David Zuhn have made significant contributions.

Overall Structure of GDB

GDB consists of three major subsystems: user interface, symbol handling (the “symbol side”), and target system handling (the “target side”):

- The user interface for most users is the command-line interface, the most common and most familiar, since it functions as a command interpreter, designed to allow for the set of commands to be augmented dynamically; it also has a recursive subcommand capability, where the first argument to a command may itself direct a lookup on a different command list. For the graphical user interface, see “Insight, the GNUPro Debugger GUI” on page 169.
- The symbol side consists of object file readers, debugging info interpreters, symbol table management, source language expression parsing, type and value printing. The symbolic side of GDB can be thought of as “everything you can do in GDB without having a live program running.” For instance, you can look at the types of variables, and evaluate many kinds of expressions.
- The target side consists of execution control, stack frame analysis, and physical target manipulation. The target side/symbol side division is not formal, and there are a number of exceptions. For instance, core file support involves symbolic elements (the basic core file reader is in BFD) and target elements (it supplies the contents of memory and the values of registers). Instead, this division is useful for understanding how the minor subsystems should fit together. The target side of GDB is the “bits and bytes manipulator.” Although it may make reference to symbolic information here and there, most of the target side will run with only a stripped executable available—or even no executable at all, in remote debugging cases.

Operations such as disassembly, stack frame crawls, and register display, are able to work with no symbolic information at all. In some cases, such as disassembly, GDB will use symbolic information to present addresses relative to symbols rather than as raw numbers, but it will work either way.

Configuring GDB

The term, *host*, refers to attributes of the system where GDB runs. The term, *target*, refers to the system where the program being debugged executes. In most cases they are the same machine, in which case a third term, *native attributes*, comes into play.

When you want to make GDB work as *native* on a particular machine, you have to include all three kinds of information.

Defines and include files needed to build on the host are host supported. Examples are tty support, system defined types, host byte order, host float format. Defines and information needed to handle the target format are target dependent. Examples are the stack frame format, instruction set, breakpoint instruction, registers, and setting up and removing the stack to call a function.

Information that is only needed when the host and target are the same, is native-dependent. One example is UNIX child process support (if the host and target are not the same, doing a fork to start the target process is a bad idea; the various macros needed for finding the registers in the `upage`, running `ptrace`, and such are all in the native-dependent files). Another example of native-dependent code is support for features that are really part of the target environment, but which require `#include` files that are only available on the host system; core file handling and `set jmp` handling are two common cases.

Symbol Handling for GDB

A key part of GDB's operation are *symbols*. Symbols include variables, functions, and types.

Symbol Reading

GDB reads symbols from symbol files. The usual symbol file is the file containing the program which GDB is debugging. GDB can be directed to use a different file for symbols (with the `symbol-file` command), and it can also read more symbols using the `add-file` and `load` commands, or while reading symbols from shared libraries.

Symbol files are initially opened by code in `symfile.c` using the BFD library. BFD identifies the type of the file by examining its header. `symfile_init` then uses this identification to locate a set of symbol-reading functions.

Symbol reading modules identify themselves to GDB by calling `add_symtab_fns` during their module initialization. The argument to `add_symtab_fns` is a struct, `sym_fns`, which contains the name (or name prefix) of the symbol format, the length of the prefix, and pointers to four functions. These functions are called at various times to process symbol-files whose identification matches the specified prefix.

The functions supplied by each module are:

```
xyz_symfile_init(struct sym_fns *sf)
```

Called from `symbol_file_add` when we are about to read a new symbol file. This function should clean up any internal state (possibly resulting from half-read previous files, for example) and prepare to read a new symbol file. The symbol

file might be a new “main” symbol file, or might be a secondary symbol file whose symbols are being added to the existing symbol table.

The argument to `xyz_symfile_init` is a newly allocated struct, `sym_fns`, whose BFD field contains the BFD for the new symbol file being read. Its private field has been zeroed, and can be modified as desired. Typically, a struct of private information will be `malloc`'d, and a pointer to it will be placed in the private field.

There is no result from `xyz_symfile_init`, but it can call error if it detects an unavoidable problem.

`xyz_new_init()`

Called from `symbol_file_add` when discarding existing symbols. This function need only handle the symbol-reading module's internal state; the symbol table data structures visible to the rest of GDB will be discarded by `symbol_file_add`. It has no arguments and no result. It may be called after `xyz_symfile_init`, if a new symbol table is being read, or may be called alone if all symbols are simply being discarded.

`xyz_symfile_read(struct sym_fns *sf, CORE_ADDR addr, int mainline)`

Called from `symbol_file_add` to actually read the symbols from a symbol-file into a set of `psymtabs` or `symtabs`.

`sf` points to the struct, `sym_fns`, originally passed to `xyz_sym_init` for possible initialization. `addr` is the offset between the file's specified start address and its true address in memory. `mainline` is 1 if this is the main symbol table being read, and 0 if a secondary symbol file is being read (that is, a shared library or dynamically loaded file).

In addition, if a symbol-reading module creates `psymtabs` when `xyz_symfile_read` is called, these `psymtabs` will contain a pointer to a `xyz_psymtab_to_symtab` function, which can be called from any point in the GDB symbol-handling code.

`xyz_psymtab_to_symtab (struct partial_symtab *pst)`

Called from `psymtab_to_symtab` (or the `PSYMTAB TO SYMTAB` macro) if the `psymtab` has not already been read in and had its `pst->symtab` pointer set.

The argument is the `psymtab` to be fleshed-out into a `symtab`. Upon return, `pst->readin` should have been set to 1, and `pst->symtab` should contain a pointer to the new corresponding `symtab`, or zero if there were no symbols in that part of the symbol file.

Partial Symbol Tables

GDB has three types of *symbol tables*.

- full symbol tables (`symtabs`), which contain the main information about symbols and addresses.
- partial symbol tables (`psymtabs`), which contain enough information to know when to read the corresponding part of the full symbol table.

- minimal symbol tables (`msymtabs`), which contain information gleaned from non-debugging symbols.

The following documentation describes partial symbol tables.

A `psymtab` is constructed by doing a very quick pass over an executable file's debugging information. Small amounts of information are extracted—enough to identify which parts of the symbol table will need to be re-read and fully digested later, when the user needs the information. The speed of this pass causes GDB to start up very quickly. Later, as the detailed rereading occurs, it occurs in small pieces, at various times, and the delay therefrom is mostly invisible to the user.

The symbols that show up in a file's `psymtab` should be, roughly, those visible to the debugger's user when the program is not running code from that file. These include external symbols and types, static symbols and types, and `enum` values declared at file scope.

The `psymtab` also contains the range of instruction addresses that the full symbol table would represent. The idea is that there are only two ways for the user (or much of the code in the debugger) to reference a symbol:

- by its address (that is, execution stops at some address which is inside a function in this file). The address will be noticed to be in the range of this `psymtab`, and the full `symtab` will be read in. `find_pc_function`, `find_pc_line`, and other `find_pc_` functions handle this.
- by its name (that is, the user asks to print a variable, or set a breakpoint on a function).

Global names and file-scope names will be found in the `psymtab`, which will cause the `symtab` to be pulled in. Local names will have to be qualified by a global name, or a file-scope name, in which case we will have already read in the `symtab` as we evaluated the qualifier. Or, a local symbol can be referenced when we are in a local scope, in which case the first case applies. `lookup_symbol` does most of the work here.

The only reason that `psymtabs` exist is to cause a `symtab` to be read in at the right moment. Any symbol that can be elided from a `psymtab`, while still causing that to happen, should not appear in it. Since `psymtabs` don't have the idea of scope, you can't put local symbols in them anyway. `psymtabs` don't have the idea of the type of a symbol, either, so types need not appear, unless they will be referenced by name.

It is a bug for GDB to behave one way when only a `psymtab` has been read, and another way if the corresponding `symtab` has been read in. Such bugs are typically caused by a `psymtab` that does not contain all the visible symbols, or which has the wrong instruction address ranges. The `psymtab` for a particular section of a symbol-file (`objfile`) could be thrown away after the `symtab` has been read in. The `symtab` should always be searched before the `psymtab`, so the `psymtab` will never be used (in a bug-free environment). Currently, `psymtabs` are allocated on an `obstack`, and all the `psymbols` themselves are allocated in a pair of large arrays on an `obstack`, so there is

little to be gained by trying to free them unless you want to do a lot more work.

Types

There are some considerations for *types*.

- Fundamental types (such as `FT_VOID`, `FT_BOOLEAN`) which GDB uses internally. Fundamental types from the various debugging formats (stabs, ELF, etc.) are mapped into one of these. They are basically a union of all fundamental types with which GDB associates languages.
- Type codes (such as `TYPE_CODE_PTR`, `TYPE_CODE_ARRAY`), marked by GDB each time that GDB builds an internal type. The type may be a *fundamental type*, such as `TYPE_CODE_INT`, or a *derived type* (a pointer to another type), such as `TYPE_CODE_PTR`. Typically, several `FT *` types map to one `TYPE_CODE *` type, and are distinguished by other members of the type struct, such as whether the type is signed or unsigned, and how many bits it uses.
- Builtin types (such as builtin type `void`, builtin type `char`), which are instances of type structs that roughly correspond to fundamental types and are created as global types for GDB to use for various ugly historical reasons. The builtin type `int` initialized in `gdbtypes.c` is basically the same as a `TYPE_CODE_INT` type that is initialized in `c-lang.c` for an `FT_INTEGER` fundamental type. The difference is that the builtin type is not associated with any particular object file, and only one instance exists, while `c-lang.c` builds as many `TYPE_CODE_INT` types as needed, with each one associated with some particular object file.

Object File Formats for GDB

The following documentation discusses the *object file formats* for GDB.

- `a.out`, the original file format for UNIX, consisting of three sections: `text`, `data`, and `bss`, which are, respectively, for program code, initialized data, and uninitialized data. The `a.out` format is so simple that it doesn't have any reserved place for debugging information. (Original UNIX hackers used `adb`, which is a machine-language debugger.) The only debugging format for `a.out` is stabs, which is encoded as a set of normal symbols with distinctive attributes. The basic `a.out` reader is in `dbxread.c`.
- COFF, a format introduced with System V Release 3 (SVR3) UNIX. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging. Although this was a step forward, the debugging information was woefully limited. For instance, it was not possible to represent code that came from an included file. The COFF reader is in `coffread.c`.

- ECOFF, an extended COFF originally introduced for MIPS and Alpha workstations. The basic ECOFF reader is in `mipsread.c`.
- XCOFF, from the IBM RS/6000 running AIX using this object file format. The COFF sections, symbols, and line numbers are used, but debugging symbols are dbx-style stabs whose strings are located in the `.debug` section (rather than the string table). GDB can only debug C++ code if you compile with the GNU C++ compiler, G++, and C++ debugging depends on the use of additional debugging information in the symbol table, thus requiring special support. GDB has this support only with the stabs debug format. In particular, if your compiler generates XCOFF with stabs extensions to the symbol table, these facilities are all available. (With GCC, 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.

The shared library scheme has a clean interface for figuring out what shared libraries are in use, but the catch is that everything referring to addresses (symbol tables and breakpoints at least) needs to be relocated for both shared libraries and the main executable. At least using the standard mechanism, this can only be done once the program has been run (or the core file has been read).

- PE, a format used by Windows 95 and NT (the Portable Executable format) for their executables. PE is basically COFF with additional headers. While BFD includes special PE support, GDB needs only the basic COFF reader.
- ELF, the format originally from the System V Release 4 (SVR4) UNIX. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. The basic ELF reader is in `elfread.c`.
- SOM, HP's object file and debug format (not to be confused with IBM's SOM, a cross-language ABI). The SOM reader is in `hpread.c`.
- Other file formats that have been supported by GDB include Netware Loadable Modules (`nlmread.c`).

Debugging File Formats

The following documentation describes characteristics of debugging information that are independent of the object file format.

- *stabs*
stabs started out as special symbols within the `a.out` format. Since then, it has been encapsulated into other file formats, such as COFF and ELF. While `dbxread.c` does some of the basic stab processing, including for encapsulated versions, `stabsread.c` does the real work.
- COFF
The basic COFF definition includes debugging information. The level of support is minimal and non-extensible, and is not often used.

- MIPS debug (Third Eye)
ECOFF includes a definition of a special debug format. The file, `mdebugread.c`, implements reading for this format.
- DWARF 1
DWARF 1 is a debugging format that was originally designed to be used with ELF in SVR4 systems. The DWARF 1 reader is in `dwarfread.c`.
- DWARF 2
DWARF 2 is an improved but incompatible version of DWARF 1. The DWARF 2 reader is in `dwarf2read.c`.
- SOM
Like COFF, the SOM definition includes debugging information.

Adding a New Symbol Reader to GDB

If you are using an existing object file format (`a.out`, COFF, ELF, etc.), there is probably little to be done.

If you need to add a new object file format, you must first add it to BFD. This is beyond the scope of this document. You must then arrange for the BFD code to provide access to the debugging symbols. Generally GDB will have to call swapping routines from BFD and a few other BFD internal routines to locate the debugging information. As much as possible, GDB should not depend on the BFD internal data structures.

For some targets (such as COFF), there is a special transfer vector used to call swapping routines. Since the external data structures on various platforms have different sizes and layouts. Specialized routines that will only ever be implemented by one object file format may be called directly. This interface should be described in a file, `bfd/libxyz.h`, which is included by GDB.

2

Installing GDB

The following documentation discusses all that is necessary for building GDB from source code, compiling it, and installing it.

IMPORTANT! You do not need this information when you have GDB already installed from the shipped distribution. This documentation is specifically for developers who are downloading the source code, compiling it themselves, and installing GDB.

- “Locating Files for Installing GDB” on page 18
- “Compiling GDB in Another Directory” on page 19
- “Specifying Names for Hosts and Targets” on page 20
- “configure Options with GDB” on page 20

Locating Files for Installing GDB

GDB comes with a `configure` script that automates the process of preparing GDB for installation; you can then use GNU `make` to build GDB. The downloadable releases include all the source code you need for GDB, which consist of the following files.

- `configure` (and supporting files)
Script for configuring GDB and all its supporting libraries
- `gdb`
Source specific to GDB
- `bfd`
Source for the Binary File Descriptor library
- `include`
GNU include files
- `libiberty`
Source for the `-liberty` free software library
- `opcodes`
Source for the library of opcode tables and disassemblers
- `readline`
Source for the GNU command-line interface
- `mmalloc`
Source for the GNU memory-mapped `malloc` package

The simplest way to configure and build GDB is to run `configure` using `gdb-version` sources in a separate build directory (see “Compiling GDB in Another Directory” on page 19).

Pass the identifier for the platform on which GDB will run as an argument. Consider the following example’s input.

```
cd gdb-version
./configure host
make
```

`host` is an identifier 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 GNU `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, `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, use the following example's input to configure only the `bfd` subdirectory:

```
cd ./bin/gdb-version-number/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.

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; for more on the `VPATH` option, see *Using make in GNUPro Development Tools*), 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 the current version, you can build GDB in a separate directory for your machine, using the following declaration (where `version` is the version which you have installed by default and `host` is the host machine with which you installed the tools).

```
cd gdb-version
mkdir ../gdb-host
cd ../gdb-host
../gdb-version/configure host
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 `host` library, `libiberty.a`, in the directory `gdb-host/libiberty`, and GDB itself in `gdb-host/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-version` (or in a separate directory configured with `--srcdir=dirname/gdb-version`), 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.

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 triplet pattern: *architecture-vendor-os*. For example, use the alias, `sun4`, as a *host* argument, or as the value for `target` in a `--target=target` option. `sparc-sun-sunos4` is the equivalent full name.

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. `config.sub` is also distributed in the GDB source directory.

configure Options with GDB

The following example summarizes the `configure` options and arguments that are most often useful for building GDB. `configure` also has several other options not listed here. See the `configure.info` file with its `What Configure Does` node for a full explanation of `configure`.

```
configure [--help]
 [--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 `dir` directory.

`--srcdir=dirname`

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*.

WARNING! Using this option requires GNU `make`, or another `make` that implements the `VPATH` feature; for more on the `VPATH` option, see *Using make* in ***GNUPro Development Tools***.

`--norecursion`

Configure only the directory level where `configure` is executed; do not propagate configuration to subdirectories.

`--rm`

Remove files otherwise built during configuration.

`--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.

`configure` accepts other options, for compatibility with configuring other GNU tools recursively; but these are the only options that affect GDB or its supporting libraries.

3

Essentials of GDB

The following documentation discusses the essentials of GDB, invoking the debugger, choosing files, choosing modes, stopping a process and shell commands.

Primarily, to start GDB and quit GDB, use the following instructions.

- Type `gdb` to start the debugger in a graphical interface mode or use the `gdb -nw` command to start the debugger in a non-window interface (command-line) mode.
- Type `quit` or use the keystroke sequence, **Ctrl-d**, to exit.

The following documentation discusses other essentials of working with GDB.

- “Invoking GDB” on page 23
- “Choosing Files for GDB to Debug” on page 24
- “Choosing Modes” on page 26
- “Quitting GDB” on page 27
- “Shell Commands for GDB” on page 27

Invoking GDB

Invoke GDB by using the command, `gdb`. Once started, GDB reads commands from the terminal until you provide a command to quit. 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 in the following discussions are

designed to cover a variety of situations; in some environments, effectively, some of these options may be unavailable.

The most usual way to start GDB is with one argument, specifying an executable program, *program*, that you want to debug.

```
gdb program
```

You can also start with both an executable program and a core file specified as the following example's input shows.

```
gdb program core
```

You can, instead, specify a process ID as a second argument, if you want to debug a running process, for instance, as the following example's input shows.

```
gdb program 1234
```

Your machine hereby attaches 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.

Run GDB without printing the front material, which describes GDB's non-warranty, using the following input:

```
gdb -silent
```

You can further control how GDB starts up by using command-line options. To display all available options and briefly describe their use, use `gdb -help` as input (`gdb -h` is a shorter equivalent).

All options and command line arguments process in sequential order. The order makes a difference when using the `-x` option.

Choosing Files for GDB to Debug

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.)

Many options have 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 this documentation provides 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 “Command Files” on page 165.

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

`-m`
`-mapped`
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/foo`, the mapped symbol file is `./foo.syms`. Most 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.

WARNING! This option depends on facilities not available or supported on all systems.

`-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.

The `-mapped` and `-readnow` options are typically combined in order to build a `.syms` file that contains complete symbol information. (See “Commands to Specify Files”

on page 125 for information.

A `.syms` file for future use is what the following example shows.

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

Choosing Modes

Run GDB in alternative modes (for example, in batch mode or quiet mode).

`-nx`

`-n`

Do not execute commands from any initialization files (normally called `.gdbinit`). Normally, the commands in these files are executed after all the command options and arguments have been processed. See “Command Files” on page 165.

`-quiet`

`-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 non-zero 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 following message does not issue when running in batch mode. Ordinarily, the message does issue whenever a program running under GDB control terminates.

```
Program exited normally.
```

`-cd directory`

Run GDB using *directory* as its working directory, instead of the current directory.

`-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. With Emacs as the GDB interface, use the two `\032` characters as a signal to display the source code for the frame.

- b *bps*
Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.
- tty *device*
Run using *device* for your program's standard input and output.
- 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.
- write
Open the executable and core files for both reading and writing. This is equivalent to the set `write on` command (see "Patching Programs" on page 122).

Quitting GDB

`quit`
To exit GDB, use the `quit` command (abbreviated `q`), or use an end-of-file character (usually **Ctrl-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, **Ctrl-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 use the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.

If you have been using GDB to control an attached process or device, you can release it with the `detach` command (see "Debugging a Running Process" on page 40).

Shell Commands for GDB

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 the standard shell to execute *command string*. If it exists, the environment variable, `SHELL`, determines which shell to run.

Otherwise GDB uses `/bin/sh`.

The `make` utility 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, `make-args` (this is equivalent to `shell make make-args`).

4

GDB Commands

The following documentation discusses GDB commands.

- “Command Syntax” (below)
- “Command Completion” on page 30
- “Getting Help” on page 32

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, which is followed by arguments whose meaning depends on the command name. For example, the command, `step`, accepts an argument which is the number of times to step, as in `step 5`. You can also use the `step` command with no arguments. Some command names do not allow any arguments.

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous. You can repeat certain GDB commands by using the **Return** (or **Enter**) key. 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).

GDB command names may always be truncated if that abbreviation is unambiguous. Other 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, using the **Return** (or **Enter**) key just once, means to repeat the previous command. Certain commands (for example, `run`) will not repeat this way; such commands have unintentional repetition which might cause trouble; because of their nature, you probably do not want to repeat such commands.

The `list` and `x` commands, when you repeat them with **Return** (or **Enter**) key actions, construct new arguments rather than repeating exactly as generated. This permits easy scanning of source or memory.

GDB can also use **Return** (or **Enter**) in another way: to partition lengthy output, in a way similar to the common utility, `more` (see “Screen Size” on page 160). Since it is easy to use **Return** (or **Enter**) one too many times 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 “Command Files” on page 165).

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, at any time, what the valid possibilities are for the next word in a command. This works for GDB commands, GDB subcommands, and the names of symbols in your program.

Use 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 use **Return** (or **Enter**) to enter it. For example, if you type `(gdb) info bre`, and use the **Tab** key, GDB fills in the rest of the word `breakpoints`, since that is the only `info` subcommand beginning with `bre`. Either use **Return** (or **Enter**) at this point, to run the `info breakpoints` command, or use the **Backspace** key 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 use **Return** (or **Enter**) 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 use the **Tab** key, either supply more characters and try again, or just use the **Tab** key a second time (GDB then 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_` and use the **Tab** key, use the **Tab** key again to display all the function names in your program that begin with those characters. For example, type `(gdb) b make_` and then use the **Tab** key; you use the **TAB** key again,

and see the following display.

```

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)

```

After displaying the available possibilities, GDB copies your partial input (input would be `b make_`) so you can finish the command. If you just want to see the list of alternatives in the first place, you can get help by using the command key sequence, **Meta-?** rather than using **Tab** twice.

IMPORTANT! **Meta-** means using the **Meta** key (the diamond key, or, alternatively, **Alt**) along with an accompanying key as a command key sequence (such as **? for help**).

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 use the **Tab** key or **Meta-?** to request word completion, as in the following example.

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

Use the **Meta-?** command key sequence this point.

```

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, as in the following example's declaration.

```
(gdb) b bub
```

Use the **Tab** key at this point. GDB alters your input line then to the following declaration, 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.

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 like the following output.

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

```
running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
```

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, the following output shows the `help` display for the class, `status`.

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

```
List of commands:
```

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

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.

`complete args`

The `complete args` command lists all the possible completions for the beginning of a command. With `args`, specify the beginning of the command you want completed; for example, output for `info`, `inspect` or `ignore`. This command is intentionally 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 subcommands (see “Index” on page 243).

`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` subcommands 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.

The following 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 occasionally want to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. The version number is also announced when you start GDB.

`show copying`

Display information about permission for copying GDB.

`show warranty`

Display the GNU “NO WARRANTY” statement.

5

Running Programs under GDB

When you run a program under GDB, you must first generate debugging information when you compile it. You may start GDB with its arguments, if any, in an environment of your choice. You may redirect your program's input and output, debug an already running process, or kill a child process.

For more discussion, see the following topics.

- “Compiling for Debugging” on page 36
- “Starting a Program” on page 36
- “Your Program's Arguments” on page 37
- “Your Program's Environment” on page 38
- “Your Program's Working Directory” on page 39
- “Your Program's Input and Output” on page 39
- “Debugging a Running Process” on page 40
- “Killing the Child Process” on page 41
- “Additional Process Information” on page 41
- “Debugging Programs with Multiple Threads” on page 42
- “Debugging Programs with Multiple Processes” on page 44

Compiling for Debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the `-g` option when you run the compiler. Many C compilers are unable to handle the `-g` and `-O` options together. Using those compilers, you cannot generate optimized executables containing debugging information.

GCC, the GNU C compiler, supports `-g` with or without the `-O` option, making it possible to debug optimized code. Always use `-g` whenever you compile a program.

When you debug a program compiled with `-g -O`, remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file; an extreme example would be defining a variable, since GDB never sees that variable because the compiler optimizes it out of existence.

Some things do not work as well with `-g -O` as with just the `-g` option, particularly on machines with instruction scheduling. If in doubt, recompile with `-g` alone, and if this fixes the problem, please report it as a bug by including a test case.

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

Starting a 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 “Essentials of GDB” on page 23), or using the `file` or `exec-file` command (see “Commands to Specify Files” on page 125).

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 certain information it receives from its superior. GDB provides ways to specify this information, which you must do *before* starting your program. (You can change it after starting your program, but such

changes only affect your program the next time you start it.) This information may be divided into the following 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. In Unix systems, you can control which shell is used with the `SHELL` environment variable. See “Your Program’s Arguments” on page 37.
- *Environment*
Your program normally inherits its environment from GDB, but you can use the GDB commands `set environment` and `unset environment` to change parts of the environment that affect your program. See “Your Program’s Environment” on page 38.
- *Working directory*
Your program inherits its working directory from GDB. You can set the GDB working directory with the `cd` command in GDB. See “Your Program’s Working Directory” on page 39.
- *Standard input and output*
Your program normally 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 “Your Program’s Input and Output” on page 39.

WARNING! While input and output redirection work, 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.

When you issue the `run` command, your program begins to execute immediately. See “Stopping and Continuing” on page 45 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 “Examining Data” on page 77.

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.

Your Program’s Arguments

The arguments to your program can be specified by the arguments of the `run` command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. Your `SHELL` environment

variable (if it exists) specifies what shell GDB uses. If you do not define `SHELL`, GDB uses `/bin/sh`.

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

`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.

Your Program's Environment

The *environment* consists of a set of environment variables and their values. Environment variables conventionally record such things as user name, home directory, terminal type, and the 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.

`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 `:` or a whitespace. 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.

`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. *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, the command, `set env USER = foo`, tells a UNIX program, when 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.

WARNING! GDB runs your program using the shell indicated by your `SHELL` environment variable if it exists (or `/bin/sh` if not). If your `SHELL` variable names a shell that runs an initialization file (such as `.cshrc` for C-shell, or `.bashrc` for BASH), any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as `.login` or `.profile`.

Your Program's 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 “Commands to Specify Files” on page 125.

`cd directory`

Set the GDB working directory to *directory*.

`pwd`

Print the GDB working directory.

Your Program's 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.

`info terminal`

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

You can redirect your program's input and/or output using shell redirection with the

`run` command. For example, `run > outfile` starts your program, diverting its output to the file `outfile`. 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.

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.

Debugging 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 use the **Return** (or **Enter**) key a second time after executing the command.

To use `attach`, your program must be running in an environment which supports processes; for example, `attach` does not work for programs on bareboard targets that lack an operating system. You must also have permission to send the process a signal.

When using `attach`, you should first use the `file` command to specify the program running in the process and load its symbol table. See “Commands to Specify Files” on page 125.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with `run`. You can insert breakpoints; you can `step` and `continue`; you can modify storage. If you would rather the process continue running, you may 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. Detaching the process continues its execution. 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 use the **Return** (or **Enter**) key 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 “Optional Warnings and Messages” on page 161).

Killing the Child Process

`kill`

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

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump files 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 use `run`, GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current breakpoint settings).

Additional Process Information

Some operating systems provide a facility called `/proc` that can be used to examine the image of a running process using file system subroutines. If GDB is configured for an operating system with this facility, the command `info proc` is available to report on several kinds of information about the process running your program. `info proc` works only on SVR4 systems that support `procfs`.

`info proc`

Summarize available information about the process.

`info proc mappings`

Report on the address ranges accessible in the program, with information on whether your program may read, write, or execute each range.

`info proc times`

Starting time, user CPU time, and system CPU time for your program and its children.

`info proc id`

Report on the process IDs related to your program: its own process ID, the ID of its parent, the process group ID, and the session ID.

`info proc status`

General information on the state of the process. If the process is stopped, this report includes the reason for stopping, and any signal received.

`info proc all`

Show all the above information about the process.

Debugging Programs with Multiple Threads

In some operating systems, 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 perhaps private memory. GDB provides the following facilities for debugging multi-thread programs.

- automatic notification of new threads
- `thread threadno`, a command to switch among threads by the thread's number (*threadno*)
- `info threads`, a command to inquire about existing threads
- `thread apply {[threadno]|[all]} args`, a command to apply to a list of threads, denoted by the thread's number (*threadno*), or to arguments (*args*)
- thread-specific breakpoints

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.

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 instance, a system without thread support shows no output from `info threads` and always rejects the `thread` command, like the following example shows.

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

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 the following output when GDB notices a new thread.

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

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 the following order):

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

An asterisk (*) to the left of the GDB thread number indicates the current thread. Use the following example for clarity.

```
(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=0x7fffffff8)
   at threadtest.c:68
```

```
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 selected thread, and its current stack frame summary, as in the following output.

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

As with the previous `[New ...]` message for GDB's output, 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 (where `systag` depends on your system's conventions).

See “Stopping and Starting Multiple Thread Programs” on page 62 for more information about how GDB behaves when you stop and start programs with multiple threads.

See “Setting Watchpoints” on page 50 for information about watchpoints in programs with multiple threads.

Debugging Programs with Multiple Processes

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 `attach` with “Debugging a Running Process” on page 40). From that point on you can debug the child process just like any other process you attached.

6

Stopping and Continuing

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and determine causes. Inside GDB, your program may stop for any of several reasons, such as at signal, a breakpoint, or a new line after using a GDB command like `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. The following documentation discusses these topics.

- “Breakpoints, Watchpoints, and Exceptions” on page 46
- “Setting Breakpoints” on page 47
- “Setting Watchpoints” on page 50
- “Setting Catchpoints” on page 51
- “Deleting Breakpoints” on page 52
- “Disabling Breakpoints” on page 53
- “Break Conditions” on page 54
- “Breakpoint Command Lists” on page 56
- “Breakpoint Menus” on page 57
- “Continuing and Stepping” on page 58
- “Signals” on page 60
- “Stopping and Starting Multiple Thread Programs” on page 62

Usually, the messages shown by GDB provide ample explanation of the status of your

program—but you can also explicitly request this information at any time. `info program` displays information about the status of your program: whether it is running or not, what process it is, and why it stopped.

Breakpoints, Watchpoints, and Exceptions

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 “Setting Breakpoints” on page 47) to specify the place where your program should stop by line number, function name or exact address in the program.

In languages with exception handling (such as GNU C++), you can also set breakpoints where an exception is raised (see “Setting Catchpoints” on page 51).

In HP-UX SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can set breakpoints in shared libraries before the executable is run. There is a minor limitation on HP-UX systems: you must wait until the executable is run in order to set breakpoints in shared library routines that are not called directly by the program (for example, routines that are arguments in a `pthread_create` call).

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see “Setting Watchpoints” on page 50), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See “Automatic Display” on page 83.

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 “Setting Catchpoints” on page 51), 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 “Signals” on page 60.)

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.

Setting Breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The `$bpnum` debugger convenience variable records the number of the breakpoints you've set most recently (see “Convenience Variables” on page 90 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 “Breakpoint Menus” on page 57 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 frame.

`break linenum`

Set a breakpoint in the current source file at line, *linenum*. That file is the last file whose source text was printed. This breakpoint stops 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 “Examining the Stack” on page 65). 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` doesn't 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 previously (or no argument) specifying where to break. See “Break Conditions” on page 54 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 “Deleting Breakpoints” on page 52.

`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. DSU 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 only take two data breakpoints, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones. See “Break Conditions” on page 54.

`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 “Disabling Breakpoints” on page 53 and “Break Conditions” on page 54.

`rbreak regex`

Set breakpoints on all functions matching regular expression, *regex*. 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.

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 and watchpoints set and not deleted, with the following place-settings 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, follow.

`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 “Examining Memory” on page 82).

`info break` now 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 “Break Conditions” on page 54). 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 `maint info breakpoints` GDB maintenance command.

```
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, as in the following clarifications.

`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.

Setting Watchpoints

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

Watchpoints currently execute two orders of magnitude more slowly than other breakpoints, but this can be well worth it to catch errors where you have no clue what part of your program is the culprit.

`watch expr`

Set a watchpoint for an expression. GDB will break when `expr` is written into by the program and its value changes. This can be used with the new trap-generation provided by SPARClite DSU. DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. For the data addresses, DSU facilitates the `watch` command. However the hardware breakpoint registers can only take two data watchpoints, and both watchpoints must be the same kind. For example, you can set two watchpoints with `watch` commands, two with `rwatch` commands, or two with `awatch` commands, but you cannot set one watchpoint with one command and the other with a different command. GDB will reject the command if you try to mix watchpoints. Delete or disable unused watchpoint commands before setting new ones.

`rwatch expr`

Set a watchpoint that will break when `watch args` is read by the program. If you use both watchpoints, both must be set with the `rwatch` command.

`awatch expr`

Set a watchpoint that will break when `args` is read and written into by the

program. If you use both watchpoints, both must be set with the `awatch` command.

`info watchpoints`

Prints a list of watchpoints, breakpoints and catchpoints; it is the same as `info break`.

If you call a function interactively using `print` or `call`, any watchpoints you have set will be inactive until GDB reaches another kind of breakpoint or the call completes.

WARNING! In multi-thread programs, watchpoints have only limited usefulness. With the current watchpoint implementation, GDB can only watch the value of an expression in a *single thread*. If you are confident that the expression can only change due to the current thread's activity (and if you are also confident that no other thread can become current), then you can use watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression.

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 calls.

`throw`

The throwing of a C++ exception.

`catch`

The catching of a C++ exception.

`exec`

A call to `exec`.

`fork`

A call to `fork`.

`vfork`

A call to `vfork`.

`load`

`load libname`

The dynamic loading of any shared library, or the loading of the library, *libname*.

`unload`

`unload libname`

The unloading of any dynamically loaded shared library, or the unloading of the library, *libname*.

`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 (see previous `catch event` calls for *event* definitions).

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, as the following discussion describes.

- 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 “Breakpoints, Watchpoints, and Exceptions” on page 46).

With a conditional breakpoint (see “Break Conditions” on page 54) 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.

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 “Selecting a Frame” on page 67). 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 designated function, *function*.

`clear linenum`

`clear filename: linenum`

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

`delete [breakpoints][bnums...]`

Delete the breakpoints, watchpoints or catchpoint of the numbers 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`.

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. You disable and enable breakpoints, watchpoints or 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 or catchpoints if you do not know which numbers to use.

A breakpoint, watchpoint or catchpoint can have 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. A breakpoint set

with the `tbreak` command starts out in this state.

- *Enabled for deletion*

The breakpoint stops your program, but immediately after it does so it is deleted permanently.

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

```
disable [breakpoints][bnums ...]
```

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][bnums ...]
```

Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

```
enable [breakpoints] once bnums...
```

Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately after stopping your program.

```
enable [breakpoints] delete bnums...
```

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 “Setting breakpoints” on page Setting Breakpoints), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the previously listed commands. (The command, `until`, can set and delete a breakpoint of its own, but it doesn’t change the state of other breakpoints; see “Continuing and stepping” on page “Continuing and Stepping” on page 58.)

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 “Expressions” on page 78). 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 a condition, `assert`, you should set the `! assert` condition on the appropriate breakpoint (where `assert` signifies the condition to assert).

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 for the purpose of performing side effects when a breakpoint is reached (see “Breakpoint Command Lists” on page 56).

Break conditions can be specified when a breakpoint is set, by using `if` in the arguments to the `break` command. See “Setting Breakpoints” on page 47 for more discussion. They can also be changed at any time with the `condition` command. The `watch` command does not recognize the `if` keyword; `condition` is the only way to impose a further condition on a watchpoint.

`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. GDB does not actually evaluate *expression* at the time the condition command is given, however. See “Expressions” on page 78.

`condition bnum`

Remove the condition from breakpoint number *bnum*. It becomes an ordinary subsequent 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 “Continuing and Stepping” on page 58.

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 can achieve the effect of the ignore count with a condition such as `$foo-- < 0` that uses a debugger convenience variable that is decremented each time. See “Convenience Variables” on page 90.

Ignore counts apply to breakpoints, watchpoints, and catchpoints.

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

Using the **Return** or **Enter** key 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.

After a command that resumes execution, any other commands in the command list 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 `echo`, `output`, and `printf` commands allow you to print precisely controlled output, and are often useful in silent breakpoints. See “Commands for Controlled Output” on page 166. For instance, the following example shows how to 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.

The following is an example.

```
break 403
commands
silent
setx=y +4
cont
end
```

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. `[0]` cancel and `[1]` all are always the first two options. 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, `foo::overloadarg`.

```
(gdb) b foo::overloadarg
[0] cancel
[1] all
[2] foo::overload1arg(double) at ovldbbreak.cc:121
[3] foo::overload1arg(float) at ovldbbreak.cc:120
[4] foo::overload1arg(unsigned long) at ovldbbreak.cc:119
...
Multiple breakpoints were set.
Use the "delete" command to delete unwanted breakpoints.
(gdb)
```

Continuing and Stepping

Continuing means resuming program execution until your program stops at a breakpoint or watchpoint, receives a signal, completes normally, or terminates abnormally. 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 due to a signal, you may want to use `handle`, or use `signal 0` to resume execution. See “Signals” on page 60.

```
continue [ignore-count]  
c [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 “Break Conditions” on page 54). 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 synonym, `c` (`continue`), is provided purely for convenience, having exactly the same behavior as `continue`.

To resume execution at a different place, you can use `return` (see “Returning from a Function” on page 122) to go back to the calling function; or `jump` (see “Continuing at a Different Address” on page 120) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see “Breakpoints, Watchpoints, and Exceptions” on page 46 for more discussion) 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 in the following discussion.

The `step` command now only stops at the first instruction of a source line. This prevents the multiple stops that used to occur in switch statements, for loops, etc.

`step` continues to stop if a function that has debugging information is called within the line.

Also, the `step` command now only enters a subroutine if there is line number information for the subroutine. Otherwise it acts like the `next` command. This avoids problems when using `cc -g1` 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 now only stops at the first instruction of a source line. This prevents the multiple stops that used to 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 “Returning from a Function” on page 122).

`u`

`until`

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 through 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 counter-intuitive results if the order of machine code does not match the order of the source lines. For instance, in the following example 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 “Setting Breakpoints” on page 47). This form of the command uses breakpoints, and hence is quicker than `until` without an argument.

```
stepi
si
```

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

It is often useful to use `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 “Automatic Display” on page 83.

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

```
nexti
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`.

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 use an interrupt (often **Ctrl-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* (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.

`info signals`

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 the new 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 use the following functionality.

`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 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 the `signal 0` command. See “Giving a Program a Signal” on page 121.

Stopping and Starting Multiple Thread Programs

When your program has multiple threads (see “Debugging Programs with Multiple Threads” on page 42), 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 the following example shows (where 28 is the *threadno*).

```
(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 operating systems, 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 use `step`. They are more likely to run when you use `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.

7

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there. The following topics have more discussion on this subject.

- “Stack Frames” on page 66
Each time your program performs a function call, information about the call is generated. That 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*. When your program stops, the GDB commands for examining the stack allow you to see all of this information. See also “Backtraces” on page 67.
- “Selecting a Frame” on page 67
One of the stack frames is *selected* by GDB and many 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 a particular frame.
- “Information about a Frame” on page 69
When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command.

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 function's 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.

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.

frame args

The `frame` command allows you to move from one stack frame to another, and to print the stack frame you select. `args` may be either the address of the frame of the stack frame number. Without an argument, `frame` prints the current stack frame.

`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`.

Backtraces

A *backtrace* is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

```
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 using the system interrupt character sequence, **Ctrl-c**.

```
backtrace n
```

```
bt n
```

Print only the innermost (*n*) frames.

```
backtrace -n
```

```
bt -n
```

Print only the outermost (*-n*) frames.

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.

The following is an example of a backtrace. It was made with the `bt 3` command, showing 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=0xf7ffb08)
        at macro.c:71
```

More stack frames would follow.

The display for frame zero (#0) 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`.

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 for selecting a stack frame; all of them finish by printing a brief description of the stack

frame just 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.

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, `frame` 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 instance, use the following as an example.

```
(gdb) up
#1  0x22f0 in main (argc=1, argv=0xf7ffbf4, env=0xf7ffbf4)
    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 “Printing Source Lines” on page 71.

```
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.

Information about a Frame

There are several other commands to print information about the selected stack frame.

```
frame
```

```
f
```

When used without any argument, 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 “Selecting a Frame” on page 67.

```
info frame
```

```
info f
```

Prints a verbose description of the selected stack frame, including the following information.

- 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 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
```

Prints 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 “Selecting a Frame” on page 67.

```
info args
```

Prints the arguments of the selected frame, each on a separate line.

```
info locals
```

Prints 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`, to see the update. See “Setting Catchpoints” on page 51.

MIPS Machines and the Function Stack

MIPS based computers use an unusual stack frame, which sometimes requires GDB to search backward in the object code to find the beginning of a function.

To improve response time (especially for embedded applications, where GDB may be restricted to a slow serial line for this search) you may want to limit the size of this search, using one of these commands:

These commands are available *only* when GDB is configured for debugging programs on MIPS processors.

```
set heuristic-fence-post limit
```

Restrict GDB to examining at most *limit* bytes in its search for the beginning of a function.

A value of 0 (the default) means there is no limit. However, except for 0, the larger the limit the more bytes `heuristic-fence-post` must search and therefore the longer it takes to run.

```
show heuristic-fence-post
```

Display the current limit.

8

Examining Source Files

GDB can print parts of your program’s source, since the debugging information recorded in the program tells GDB what source files were used to build it. See the following documentation for more discussion on these subjects.

- “Printing Source Lines” on page 71
- “Searching Source Files” on page 73
- “Specifying Source Directories” on page 74
- “Source and Machine Code” on page 75

When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see “Selecting a Frame” on page 67), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source; see “Using GDB under GNU Emacs” on page 211 for details of using Emacs with GDB.

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 `list` command's following forms are most commonly used.

`list linenum`

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

`list function`

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

`list`

Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see “Examining the Stack” on page 65), this prints lines centered around that line.

`list -`

Print lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the `list` command. You can change this functionality by using `set listsize` as the following discussion describes.

`set listsize count`

Make the `list` command display *count* source lines (unless the `list` argument explicitly specifies some other number).

`show listsize`

Display the number of lines that `list` prints.

IMPORTANT! Repeating a `list` command using the **Return** or **Enter** key discards the argument, so it is equivalent to typing `list`. This is more useful than listing the same lines again. An exception is made for an argument of `-` (which is preserved in repetition so that each repetition moves up in the source file).

`list`

Supplies zero, one or two *linespec* (specifying source lines); there are several ways of writing them but the effect is always to specify some source line.

The following description discusses the possible arguments for `list`.

`list linespec`

Print lines centered around the line specified by *linespec*.

`list first,last`

Print lines from *first* to *last*. Both arguments specify source lines.

`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.

The following arguments are the ways of specifying a single source line—all the kinds of *linespec* (for specifying source lines); there are several ways of writing them but the effect is always to specify some source line.

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 instance, 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*, which may be any expression.

Searching Source Files

There are two commands for searching through the current source file for a regular expression (*regexp*).

`forward-search regexp`

`search regexp`

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

Use the `search regexp` synonym or abbreviate the command name as `fo`.

`reverse-search regexp`

The `reverse-search regexp` command checks each line, starting with the one

before the last line listed and going backward, for a match for *regexp*. It lists the line that is found. You can abbreviate this command as *rev*.

Specifying Source Directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. The directories could be moved between the compilation and your debugging session when the executables do record the names. GDB has a list of directories (*source path*) to search for source files. Each time GDB wants 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. 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 source files are found and where each line is in the file.

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

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

Add directory, *dirname*, to the front of the source path. Several directory names may be given to this command, separated by `:` or whitespace. You may specify a directory that is already in the source path; this moves it forward, so GDB searches it sooner.

You can use the `$cdir` string to refer to the compilation directory (if one is recorded), and `$cwd` to refer to the current working directory. `$cwd` tracks the current working directory as it changes during your GDB session, while `.` 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.

```
show directories
```

Print the source path; show which directories it contains.

If your source path is cluttered with directories that are no longer of interest, GDB may sometimes cause confusion by finding the wrong versions of source. You can correct the situation with the following method.

1. Use *directory* with no argument to reset the source path to empty.

- Use `directory` with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

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 now 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`. Specify source lines in any of the ways understood by the `list` command (see “Printing Source Lines” on page 71).

For instance, in the following example, `info line` discovered the location of the object code for the first line of a function, `m4_changequote`.

```
(gdb) info line m4_changequote
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350
```

Also inquire (using `*addras`, the form for `linespec`) what source line covers a particular address, as in the following 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 “Examining Memory” on page 82). Also, this address is saved as the value of the convenience variable, `$_` (see “Convenience Variables” on page 90).

```
disassemble
```

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 0x63e4 0x6404
Dump of assembler code from 0x63e4 to 0x6404:
0x63e4 <builtin_init+5340>:    ble 0x63f8 <builtin_init+5360>
0x63e8 <builtin_init+5344>:    sethi %hi(0x4c00), %o0
0x63ec <builtin_init+5348>:    ld [%i1+4], %o0
0x63f0 <builtin_init+5352>:    b 0x63fc <builtin_init+5364>
0x63f4 <builtin_init+5356>:    ld [%o0+4], %o0
```

```
0x63f8 <builtin_init+5360>:    or %o0, 0x1a4, %o0
0x63fc <builtin_init+5364>:    call 0x9288 <path_search>
0x6400 <builtin_init+5368>:    nop
End of assembler dump.
```

set assembly-language *instruction-set*

Selects the instruction set to use when disassembling the program using the `disassemble` or `x/i` commands. It is useful for architectures that have more than one native instruction set. Currently, it is only defined for the Intel x86 family.

You can set *instruction-set* to either `i386` or `i8086`; `i386` is the default.

9

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 in which your program is written (see “Using GDB with Different Languages” on page 95).

```
print exp
print /f exp
```

`exp` is an expression (in the source language). By default the value of `exp` 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 “Output Formats” on page 81.

```
print
print /f
```

If you omit `exp`, GDB displays the last value again (from the *value history*; see “Value History” on page 89) so that you can 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 “Examining Memory” on page 82.

If you are interested in information about types, or about how the fields of a struct or class are declared, use the `pptype exp` command rather than `print`. See “Examining the Symbol Table” on page 115.

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 using the syntax, `element, element ...`; for example, use the command, `print {1 2 3}` to build up an array in memory that is memory allocated in the target program.

IMPORTANT! Because C is so widespread, most of the expressions shown in examples in this documentation are in C. See “Using GDB with Different Languages” on page 95 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. See also the introduction to “Examining Data” on page 77.

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 the following operators, in addition to those common to programming languages.

@

Binary operator for treating parts of memory as arrays. See “Artificial Arrays” on page 80 for more information.

::

Allows for specifying a variable in terms of the file or function where it is defined. See “Program Variables” on page 78 for more information.

{*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*.

Program Variables

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

Variables in expressions are understood in the selected stack frame (see “Selecting a Frame” on page 67); they must be either *global* (sometimes referred to as *file-static*) or they must be *visible* (according to the scope rules of the programming language

from the point of execution in that frame). Consider the following function example.

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

This example shows that you can examine and use the variable, `a`, whenever your program is executing within the function, `foo`; however, you can only use or examine the variable, `b`, while your program is executing inside the block where `b` is declared. There is an exception; 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 (`::`) as in the following example.

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

In the previous example, *file* or *function* refer to the name of the context for the static input, *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`, use `p 'f2.c'::x` as input.

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 exiting. You may see this problem when you are stepping by machine instructions, 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; 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.

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 holding the first element, and so on.

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 “Value History” on page 89), 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)` in the following example shows.

```
(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 as a counter in an expression that prints the first interesting value, and then repeat that expression using **Return** or **Enter** keys (see “Convenience Variables” on page 90). 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. The following is an example of what you might input, after which you would use the **Return** or **Enter** keys twice.

```
set $i = 0
p dtab[$i++]>->fv
```

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. For example, to print the program counter in hex (see “Registers” on page 91), type `p/x $PC`; no space is required before the slash 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. 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*. `t` stands for two.
- 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.

* `b` cannot be used because these format letters are also used with the `x` command, where `b` stands for *byte*; see “Examining Memory” on page 82.

Examining Memory

You can use the `x` command (for “*examine*”) to examine memory in any of several formats, independently of your program’s 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 forward slash. Several commands set convenient defaults for `addr`.

`n`, the *repeat count*

The repeat count is a decimal integer; 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 uses any of the following variables.

`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. See “Expressions” on page 78 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 the `0x54320` address. `x/4xw $sp` prints the four words (w) of memory above the stack pointer in hexadecimal (x); the stack pointer is `$sp`. For more information, see “Registers” on page 91.

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 `4xw` and `4wx` output specifications mean exactly the same thing. However, the count `n` must come first; `wx4` does not work.

Even though the `u` unit size is ignored for the `s` and `i` formats, 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 “Source and Machine Code” on page 75.

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 **Return** or **Enter** keys 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.

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 the following.

```
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 exp`

Add the expression, `exp`, to the list of expressions to display each time your program stops. See “Expressions” on page 78.

`display` does not repeat if you use the **Return** or **Enter** keys again after using it.

`display/fmt exp`

For `fmt` specifying only a display format and not a size or count, add the expression `exp` to the auto-display list but arrange to display it each time in the specified format, `fmt`. See “Output Formats” on page 81.

`display/fmt addr`

For `fmt`, `i` or `s` (which can include a unit-size or a number of units, add the `addr` expression as a memory address to be examined each time your program stops. Examining means in effect using `x/fmt addr`; for more information, see “Examining Memory” on page 82. For example, `display/i $pc` can be helpful to see the machine instruction about to be executed each time execution stops (`$pc` is a common name for the program counter; see “Registers” on page 91).

`undisplay dnums ...`

`delete display dnums ...`

Remove item numbers `dnums` from the list of expressions to `display`. `undisplay` does not repeat if you use **Return** or **Enter** keys 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 once again.

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, the following 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, the following 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 earlier 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.

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 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
```

Display the symbolic form of an address if the offset between the closest earlier 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*, which interprets the address in symbolic form. For instance, the following example's input 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 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. 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.

```
set print pretty on
```

Cause GDB to print structures in an indented format with one member per line, like the following example's output.

```
$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 the following example.

```
$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.

The following 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. `on` is the default.

```
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. The choices for `style` are currently:

```
auto
```

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

```
gnu
```

Decode based on the GNU C++ compiler (`g++`) encoding algorithm. This is the default.

```
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 *Annotated C++ Reference Manual* (Margaret A. Ellis & Bjarne Stroustrup, Addison Wesley, 1990).

WARNING! This setting alone is not sufficient to allow debugging `cfront`-generated executables. GDB would require further enhancement to permit that functionality.

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.

```
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.

Value History

Values printed by the `print` command are saved in the GDB *value history*, allowing 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 successive integers starting with one. `print` shows you the history number assigned to a value by printing `$num=` before the value, where `num` is the history number.

To refer to any previous value, use `$` followed by the value's history number. The way `print` labels its output is designed to remind you of this. Just `$` 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 `p *$.next`.

You can print successive links in the chain by repeating this command, using the **Return** or **Enter** keys.

IMPORTANT! The history records values, not expressions. Consider, for instance, if the value of `x` is 4 and you type the following example's 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 just after the values last printed. If no more values are available, `show values +` produces no display.

Using the **Return** or **Enter** keys to repeat `show values n` has exactly the same effect as `show values +` as input.

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 `$` and any name preceded by `$` can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see “Registers” on page 91). Value history references, in contrast, are *numbers* preceded by `$`. See “Value History” on page 89.

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 con`.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For instance, to print a field from successive elements of an array of structures, use the following as an example.

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

Repeat that command by using the **Return** or **Enter** keys.

The following convenience variables are created automatically by GDB and given values likely to be useful.

`$_`

The variable, `$_`, is automatically set by the `x` command to the last address examined (see “Examining Memory” on page 82). Other commands which provide a default address for `x` to examine also set `$_` 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.

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 see 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*. 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[†] 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`.

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 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 “Selecting a Frame” on page 67). 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.

[†] 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 the **Return** or **Enter** keys; see also “Returning from a Function” on page 122.

```
set rstack_high_address address
```

On AMD 29K family processors, registers are saved in a separate *register stack*. There is no way for GDB to determine the extent of this stack. Normally, GDB just assumes that the stack is large enough that GDB's memory references always exist. If necessary, you can get around this problem by specifying the ending address of the register stack with the `set rstack_high_address` command. The argument should be an address, which you probably want to precede with `0x` to specify in hexadecimal.

```
show rstack_high_address
```

Display the current limit of the register stack, on AMD 29000 family processors.

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 ARM and x86 machines.

10

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 using `*p`, but in Modula-2, it is accomplished by using `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 previous in your program's native language, and allowing GDB to output values in a manner consistent with the syntax of your program's native language. The language you use to build expressions is called the *working language*.

The following documentation provides more discussion on language-specific issues.

- “Switching between Source Languages” on page 96
- “Displaying the Language” on page 97
- “Type and Range Checking” on page 98
- “Supported languages” on page 101

Switching between Source Languages

There are two ways to control the working language—either have GDB set it automatically, or select it manually yourself. You can use the `set language` command for either purpose. On startup, GDB defaults to setting the language automatically. The working language is used to determine how expressions you type are interpreted, how values are printed, and so forth. The following discussions address the source language usage.

- “List of Filename Extensions and Languages” on page 96
- “Setting GDB’s Working Language” on page 97
- “Having GDB Infer the Source Language” on page 97

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. This is most commonly a 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.

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.

```
.mod  
  Modula-2 source file  
.c  
  C source file  
.C  
.cc  
.cxx  
.cpp  
.cp  
.c++  
  C++ source file
```

```
.ch  
.c186  
.c286
```

CHILL source file.

```
.s  
.S
```

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

Setting GDB's 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 `set language lang` command, where *lang* is the name of a language, such as `c` or `modula-2`. For a list of the supported languages, use the `set language` command.

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.

Having GDB Infer the Source Language

To have GDB set the working language automatically, use the `set language local` or `set language auto` commands. 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.

Displaying the Language

The following commands help you find out which language is the working language,

and also what language in which source files were written.

`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 “Information about a Frame” on page 69 to identify the other information about the language in the source files.

`info source`

Display the source language of this source file. See “Examining the Symbol Table” on page 115 to identify the other information about the language in the source files.

Type and Range Checking

Some languages are designed to guard against you making seemingly common errors through a series of compile-time 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 a program’s correctness once it has been compiled by eliminating type mismatches, and providing active checks for range errors when your program is running. For more details, see “An Overview of Type Checking” on page 99 and “An Overview of Range Checking” on page 100.

GDB can check for conditions. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation, using the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your program’s source language. See “Supported languages” on page 101 for the default settings of supported languages.

warning! In some cases, the GDB commands for type and range checking are included and do not yet have any effect. The following discussion documents the intent of such commands.

An Overview of Type Checking

Some languages, such as Modula-2, 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 ever causing any run-time problems. Consider the following examples.

```
1 + 2 → 3
```

Compare with the following example.

```
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, but evaluate the expression anyway. When you choose the last of these, GDB evaluates expressions like the second example, 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 which make little sense to evaluate anyway. Each language defines to what degree it is strict about type. For instance, both Modula-2 and C require 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 “Supported languages” on page 101 for further details on specific languages.

GDB provides the following additional commands for controlling the type checker.

```
set check type auto
```

Set type checking on or off based on the current working language. See “Supported languages” on page 101 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 typechecking 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 current setting of type checker, and whether GDB sets it automatically.

An Overview of Range Checking

In some languages (such as Modula-2), 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 the following input is congruent.

```
m + 1 → s
```

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. See “Supported languages” on page 101 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 “Supported languages” on page 101 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 language default. If a range error occurs, 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.

Supported languages

GDB 5 supports C, C++, and Modula-2. Some GDB features may be used in expressions regardless of the language you use: the GDB `@` and `::` operators, and the `{type}addr` construct (see “Expressions” on page 78) can be used with the constructs of any supported language. The following documentation details to what degree each source language is supported by GDB. 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. There are many good books written on each of these languages; feel free to use them as a language reference or tutorial in addition to these discussions.

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 GNU C++ compiler, G++, and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with G++.

For best results when debugging C++ programs, use the `stabs` debugging format. You can select that format explicitly with the G++ command-line options (`-gstabs` or `-gstabs+`). See “Options Controlling Debugging” on page 45 in *Using GNU CC* in *GNUPro Compiler Tools* for more information.

C and C++ Operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers and 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`; and `enum`.
- *floating point types* include `float` and `double`.
- *Pointer types* include all types defined as `(type*)`.
- *Scalar types* include all of the previous types.

The following operators are supported (and listed in order of their increase in 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 $a \text{ } op = b$, and translated to $a = a \text{ } op b$. *op*= and = have the same precedence. *op* is any one of the |, ^, &, <<, >>, +, -, *, /, or % 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 “Expressions” on page 78).
 - +
-
+ (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 variable's value 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 use of & beyond what is allowed in the C++ language itself; you can use <code>&(&ref)</code> (or, if you prefer, <code>&&ref</code>) to examine the address where a C++ reference variable (declared with <code>&ref</code>) 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 <code>struct</code> and <code>union</code> data.
[]	Array indexing. <code>a[i]</code> is defined as <code>*(a+i)</code> . Same precedence as <code>-></code> .
()	Function parameter list. Same precedence as <code>-></code> .
::	C++ scope resolution operator. Defined on <code>struct</code> , <code>union</code> , and <code>class</code> types. Doubled colons also represent the GDB scope operator (see “Expressions” on page 78) with the same precedence as the C++ scope resolution operator.

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 (zero), and hexadecimal constants by a leading 0x or 0X. Constants may also end with l to specify that the constant should be treated as an `unsigned` value, or both ul or lu to specify that the constant should be treated as an `unsigned 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 another sequence of digits. The `+` is optional for positive exponents.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (`'`), or a number—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. You can also use `\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 (`" "`).
- 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 `{` and `}` braces; 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.

C++ Expressions

GDB expression handling can interpret most C++ expressions.

WARNING! GDB can only debug C++ code if you compile with the GNU C++ compiler, G++, and certain other compilers. Moreover, C++ debugging depends on the use of additional debugging information in the symbol table, and thus requires special support. GDB has this support only with the stabs debug format. 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 GCC, 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, some of the C++ support in GDB does not work.

- Member function calls are allowed; you can use expressions like the following input.

```
count = aml->GetOriginal(x, y)
```

- 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++.
- You can call overloaded functions; GDB resolves the function call to the right definition, with one restriction—you must use arguments of the type required by the function that you want to call. GDB does not perform conversions requiring constructors or user-defined type operators.
- 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 input the `set print address off` command.
- GDB supports the C++ name resolution operator `::` and 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 something like a `scope1::scope2::name` expression. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see “Program Variables” on page 78 for more details).

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`, and when GDB enters code compiled from one of these files, it sets the working language to C or C++. See “Having GDB Infer the Source Language” on page 97 for more details.

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 variables type 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.

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, `{...}` appears.

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See “Expressions” on page 78.

GDB Features for C++

Some GDB commands are particularly useful with C++, and some are designed specifically for use with C++. For instance, when you want a breakpoint in a function whose name is overloaded, GDB breakpoint menus help you specify which function definition you want; see also “Breakpoint Menus” on page 57.

The following summary discusses the commands.

`rbreak regex`

Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. See “Setting Breakpoints” on page 47.

`catch throw`
`catch catch`

Debug C++ exception handling using these commands. See “Setting Catchpoints” on page 51.

`ptype typename`

Print inheritance relationships as well as other information for type, *typename*. See “Examining the Symbol Table” on page 115.

`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 “Print Settings” on page 85.

`set print object`
`show print object`

Choose whether to print derived (actual) or declared types of objects. See “Print Settings” on page 85.

`set print vtbl`
`show print vtbl`

Control the format for printing virtual function tables. See “Print Settings” on page 85.

You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++; use `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 “Command Completion” on page 30 for more details.

Modula-2

The extensions made to GDB to support Modula-2 only support output from the GNU Modula-2 compiler (which is currently in development). Other Modula-2 compilers are not currently supported, and attempting to debug executables produced by them is most likely to give an error as GDB reads in the executable’s symbol table.

Modula-2 Operators

Operators must be defined on values of specific types. For instance, + is defined on numbers and not on structures. Operators are often defined on groups of types. For the purposes of Modula-2, the following definitions hold.

- *Integral* types consist of `INTEGER`, `CARDINAL`, and their subranges.
- *Character* types consist of `CHAR` and its subranges.
- *Floating point* types consist of `REAL`.
- *Pointer* types consist of anything declared as `POINTER TO type`.
- *Scalar* types consist of all of the previous types.
- *Set* types consist of `SET` and `BITSET` types.
- *Boolean* types consist of `BOOLEAN`.

The following operators are supported (and appear in order of their increase in precedence).

- ,
- Function argument or array index separator.
- :=
- Assignment. The value of `var :=value` is `value`.
- <
- >
- < (less than), > (greater than), for integral, floating point, or enumerated types.
- <=
- >=
- <= (less than or equal to), >= (greater than or equal to), for integral, floating point and enumerated types, or set inclusion on set types. Same precedence as < (less than).
- =
- <>
- #
- = (equality), <> or # (two ways of expressing inequality), valid on scalar types.

- Same precedence as `<`. In GDB scripts, only `<>` is available for inequality, since `#` conflicts with the script comment character.
- IN
Set membership. Defined on set types and the types of their members. Same precedence as `<`.
- OR
Boolean disjunction. Defined on boolean types.
- AND
&
Boolean conjunction. Defined on boolean types.
- @
The GDB “artificial array” operator (see “Expressions” on page 78).
- +
-
Addition and subtraction on integral and floating point types, or union and difference on set types.
- *
/
- Multiplication on integral and floating point types, or set intersection on set types.
- Division on floating point types, or symmetric set difference on set types. Same precedence as `*`.
- DIV
MOD
Integer division and remainder. Defined on integral types. Same precedence as `*`.
- ^
- Negative. Defined on `INTEGER` and `REAL` data.
- Pointer dereferencing. Defined on pointer types.
- NOT
^
- Boolean negation. Defined on boolean types. Same precedence as `^`.
- .
- `RECORD` field selector. Defined on `RECORD` data. Same precedence as `^`.
- []
- Array indexing. Defined on `ARRAY` data. Same precedence as `^`.
- ()
- Procedure argument list. Defined on `PROCEDURE` objects. Same precedence as `^`.
- ::
- .
- GDB and Modula-2 scope operators.

WARNING! Sets and their operations are not yet supported, so GDB treats the use of the operator, `IN`, or the use of operators, `+`, `-`, `*`, `/`, `=`, `<>`, `#`, `<=`, and `>=` on sets as an error.

Modula-2 Built-in Functions and Procedures

Modula-2 also makes available several built-in procedures and functions. In describing these functions and procedures, the following meta-variables are used:

- a
Represents an `ARRAY` variable.
- c
Represents a `CHAR` constant or variable.
- i
Represents a variable or constant of integral type.
- m
Represents an identifier that belongs to a set. Generally used in the same function with the metavariable, `s`. The type of `s` should be `SET OF mtype` (where `mtype` is the type of `m`).
- n
Represents a variable or constant of integral or floating point type.
- r
Represents a variable or constant of floating point type.
- t
Represents a type.
- v
Represents a variable.
- x
Represents a variable or constant of one of many types. See the explanation of the function for details.

All Modula-2 built-in procedures also return a result, discussed by the following descriptions.

`ABS(n)`

Returns the absolute value of *n*.

`CAP(c)`

If *c* is a lower case letter, it returns its upper case equivalent, otherwise it returns its argument

`CHR(i)`

Returns the character whose ordinal value is *i*.

`DEC(v)`

Decrements the value in the variable *v*. Returns the new value.

`DEC(v, i)`

Decrements the value in the variable *v* by *i*. Returns the new value.

`EXCL(m, s)`

Removes the element *m* from the set *s*. Returns the new set.

`FLOAT(i)`

Returns the floating point equivalent of the integer *i*.

`HIGH(a)`

Returns the index of the last member of *a*.

`INC(v)`

Increments the value in the variable *v*. Returns the new value.

`INC(v, i)`

Increments the value in the variable *v* by *i*. Returns the new value.

`INCL(m, s)`

Adds the element *m* to the set *s* if it is not already there. Returns the new set.

`MAX(t)`

Returns the maximum value of the type *t*.

`MIN(t)`

Returns the minimum value of the type *t*.

`ODD(i)`

Returns boolean `TRUE` if *i* is an odd number.

`ORD(x)`

Returns the ordinal value of its argument. For example, the ordinal value of a character is its ASCII value (on machines supporting the ASCII character set). *x* must be of an ordered type, which include integral, character and enumerated types.

`SIZE(x)`

Returns the size of its argument. *x* can be a variable or a type.

`TRUNC(x)`

Returns the integral part of *x*.

`VAL(t, i)`

Returns the member of the type *t* whose ordinal value is *i*.

WARNING! Sets and their operations are not yet supported, so GDB treats the use of `INCL` and `EXCL` procedures as an error.

Modula-2 Constants

GDB allows you to express the constants of Modula-2 in the following ways.

- Integer constants are simply a sequence of digits. When used in an expression, a constant is interpreted to be type-compatible with the rest of the expression. Hexadecimal integers are specified by a trailing `H`, and octal integers by a trailing `B`.

- Floating point constants appear as a sequence of digits, followed by a decimal point and another sequence of digits. An optional exponent can then be specified, in the form $E[+|-]nnn$, where $[+|-]nnn$ is the desired exponent. All of the digits of the floating point constant must be valid decimal (base 10) digits.
- Character constants consist of a single character enclosed by a pair of like quotes, either single (') or double ("). They may also be expressed by their ordinal value (their ASCII value, usually) followed by a `c`.
- String constants consist of a sequence of characters enclosed by a pair of like quotes, either single (') or double ("). Escape sequences in the style of C are also allowed. See “C and C++ Constants” on page 103 for an explanation of escape sequences.
- Enumerated constants consist of an enumerated identifier.
- Boolean constants consist of the identifiers `TRUE` and `FALSE`.
- Pointer constants consist of integral values only.
- Set constants are not yet supported.

Modula-2 Defaults

If type and range checking are set automatically by GDB, they both default to on whenever the working language changes to Modula-2. This happens regardless of whether you, or GDB, selected the working language. If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with `.mod` sets the working language to Modula-2. See “Setting GDB’s Working Language” on page 97 for further details.

Deviations from Standard Modula-2

A few changes have been made to make Modula-2 programs easier to debug. This is done primarily by loosening its type strictness.

- Unlike in standard Modula-2, pointer constants can be formed by integers. This allows you to modify pointer variables during debugging. (In standard Modula-2, the actual address contained in a pointer variable is hidden from you; it can only be modified through direct assignment to another pointer variable or expression that returned a pointer.)
- C escape sequences can be used in strings and characters to represent non-printable characters. GDB prints out strings with these escape sequences embedded. Single non-printable characters are printed using the `CHR(nnn)` format.
- The assignment operator (`:=`) returns the value of its right-hand argument.
- All built-in procedures both modify *and* return their argument.

Modula-2 Type and Range Checks

WARNING! In this release, GDB does not yet perform type or range checking.

GDB considers two Modula-2 variables type equivalent if the following conditions apply.

- They are of types that have been declared equivalent, using a `TYPE t1-t2` statement.
- They have been declared on the same line.

NOTE: This is true of the GNU Modula-2 compiler, but it may not be true of other compilers.) As long as type checking is enabled, any attempt to combine variables whose types are not equivalent is an error. Range checking is done on all mathematical operations, assignment, array index bounds, and all built-in functions and procedures.

Modula-2 Scope Operator (.) and GDB Scope Operator (::)

There are a few subtle differences between the Modula-2 scope operator (.) and the GDB scope operator (::). The two have similar syntax, as in the following example.

```
module . id
scope :: id
```

scope is the name of a module or a procedure. *module* is the name of a module; *id* is any declared identifier within your program, except another module. Using the :: operator makes GDB search the scope, *scope*, for the identifier, *id*. If it is not found in the specified *scope*, then GDB searches all *scope* occurrences, enclosing the one specified by *scope*.

Using the Modula-2 operator (.) makes GDB search the current scope for the identifier, *id*, which was imported from the definition module, *module*. With this operator, it is an error if the identifier, *id*, was not imported from definition module, *module*, or if *id* is not an identifier in *module*.

GDB and Modula-2

Some GDB commands have little use when debugging Modula-2 programs. Five subcommands of `set print` and `show print` apply specifically to C and C++: `vtbl`, `demangle`, `asm-demangle`, `object`, and `union`. The first four apply to C++, and the last to the C `union` type, which has no direct analogue in Modula-2.

The @ operator (see “Expressions” on page 78), while available while using any language, is not useful with Modula-2. Its intent is to aid the debugging of *dynamic arrays*, which cannot be created in Modula-2 as they can in C or C++. However,

because an address can be specified by an integral constant, the `{type}adrexp` construct is still useful (see “Expressions” on page 78).

In GDB scripts, the Modula-2 inequality operator, `#`, is interpreted as the beginning of a comment. Use `<>` instead.

11

Examining the Symbol Table

The commands described in this section 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 “Choosing Files for GDB to Debug” on page 24), or by one of the file management commands (see “Command Files” on page 165).

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 “Program Variables” on page 78). 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`, `.`, and `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 stackframe offset at which the variable is always stored.

IMPORTANT! `info address symbol` differs with `print &symbol`. For a register variable, `print &symbol` does not work; for a stack local variable, `info address symbol` prints information about the address while `print &symbol` prints the exact address of the current instantiation of

the variable.

`whatis exp`

Print the data type of expression, *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See “Expressions” on page 78.

`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 exp`

`ptype`

Print a description of the type of expression, *exp*. `ptype` differs from `whatis` by printing a detailed description, instead of just the name of the type. For instance, consider the following variable declaration example.

```
struct complex {double real; double imag;} v;
```

The declaration’s two commands would have the following display on your shell’s window.

```
(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 name matches *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 name includes 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 (the source file for the function containing the current point of execution) and the language in which it was written.

```
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 (except for local variables).

```
info variables regexp
```

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

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 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 the following 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 re-encountering object files of the same name. This is the default state; if you are not running on a system that permits automatically relinking 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.

```
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. Use the `info sources` command 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 “Commands to Specify Files” on page 125 for a discussion of how GDB reads symbols (in the description of `symbol-file`).

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.

For more information, see the following documentation.

- “Assignment to Variables” (below)
- “Continuing at a Different Address” on page 120
- “Giving a Program a Signal” on page 121
- “Returning from a Function” on page 122
- “Calling Program Functions” on page 122
- “Patching Programs” on page 122

Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. See “Expressions” on page 78. 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

“Using GDB with Different Languages” on page 95 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 “Value History” on page 89). The expression is evaluated only for its effects.

If the beginning of the argument string of the `set` command appears identical to a `set` subcommand, use the `set variable` command instead of only `set`. This command is identical to `set` except for its lack of subcommands. For example, if your program has a variable, `width`, you get an error if you try to set a new value with just `set width=13`, because GDB has the `set width` command:

```
(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 `(gdb) set var width=47`.

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 “Expressions” on page 78). 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.

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 instead continue at an address of your own choosing, with the following commands.

`jump linespec`

Resume execution at a specified line, `linespec`. Execution stops again immediately if there is a breakpoint there. See “Printing Source Lines” on page 71 for a description of the different forms of `linespec`. It is common practice to use the `tbreak` command in conjunction with `jump`. See “Setting Breakpoints” on page 47.

The `jump` command does not change the current stack frame, or the stack pointer,

or 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*.

You can get much the same effect as the `jump` command by storing a new value into the register, `$pc`. The difference is that this does not start your program running; it only changes the address of where it will run when you continue. 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 “Continuing and Stepping” on page 58.

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.

Giving a Program a Signal

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 “Signals” on page 60). The `signal` command passes the signal directly to your program.

`signal signal`

Resumes execution where your program stopped, but immediately gives it the signal, *signal*, which 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 ordinarily see the signal when resumed with the `continue` command; `signal 0` causes it to resume without a signal.

`signal` does not repeat when you use **Return** or **Enter** a second time after executing the command.

Returning from a Function

Use `return` so that 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`.

```
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 function's return value.

This pops the selected stack frame (see “Selecting a Frame” on page 67), 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 “Continuing and Stepping” on page 58) resumes execution until the selected stack frame returns naturally.

Calling Program Functions

Use `call` as a 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.

```
call expr
```

Evaluate the expression, *expr*, without displaying `void` returned values.

The 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.

Patching Programs

By default, GDB opens the file containing your program's executable code (or the corefile) as read-only. This prevents accidental alterations to machine code; but it also prevents you from intentionally patching your program's binary.

If you'd like to be able to patch the binary, 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 a `set write on` command, GDB opens executable and core files for both reading and writing; if you specify `set write off` (the default), GDB opens them read-only. If you have already loaded a file, you must load it again (using the `exec-file` or `core-file` commands) 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.

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.

The following documentation discusses more of GDB files.

- “Commands to Specify Files” (below)
- “Errors Reading Symbol Files” on page 129

Commands to Specify Files

You may want to specify executable and core dump file names. The usual way to do this is at start-up time, using the arguments to GDB’s start-up commands (see “Essentials of GDB” on page 23).

Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify a file you want to use. 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, `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 following descriptions of the file options, `-mapped` and `-readnow` with the commands, `file`, `symbol-file`, or `add-symbol-file`, described in the following text), 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 your program's symbol table. 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 use **Return** or **Enter** 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. Best results are usually obtained from GNU compilers; for example, using `gcc` you can generate debugging information for optimized code.

On some 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 “Optional Warnings and

Messages” on page 161.)

We have not implemented the two-stage strategy for COFF yet. When the symbol table is stored in COFF format, `symbol-file` reads the symbol table data in full right away.

```
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.

IMPORTANT! 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 “Killing the Child Process” on page 41).

```
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 a “You can’t do that when your target is...” error message.

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.

On VxWorks, `load` links *filename* dynamically on the current target system as well as adding its symbols in GDB.

With the Nindy interface to an Intel 960 board, `load` downloads *filename* to the 960 as well as adding its symbols in GDB.

When you select remote debugging to a Hitachi SH, H8/300, or H8/500 board (see “GDB and Hitachi Microprocessors” on page 152), the `load` command downloads your program to the Hitachi board and also opens it as the current executable target for GDB on your host (like the `file` command).

`load` does not repeat if you use **Return** or **Enter** again after using it.

```
add-symbol-file filename address
```

```
add-symbol-file filename address[-readnow][--mapped]
```

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 *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 command, `add-symbol-file`, 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.

`add-symbol-file` does not repeat if, after using it, you use **Return** or **Enter**

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*.

```
add-shared-symbol-file
```

The `add-shared-symbol-file` command can be used only under Harris’ CXUX operating system for the Motorola 88k. GDB automatically looks for shared libraries; however if GDB does not find yours, you can run

```
add-shared-symbol-file. It takes no arguments.
```

```
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 lists all the sections and their addresses.

```
info files
info target
```

`info files` and `info target` are synonymous; both print the current target (see “Specifying a Debugging Target” on page 131), including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The `help target` command 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 supports SunOS, SVr4, Irix 5, and IBM RS/6000 shared libraries. 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, remember that GDB does not understand references to a function in a shared library, *unless* you are debugging a core file).

```
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 using `run`. If *regex* is omitted, all shared libraries required by your program are loaded.

Errors Reading Symbol Files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known errors 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 “Optional Warnings and Messages” on page 161).

The following documentation discusses error messages and their meanings.

```
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 using the `set verbose on` command. See “Optional Warnings and Messages” on page 161.

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, `f00`, which may cause other problems if many symbols end up with this name.

unknown symbol type `0xnn`

The symbol information contains new data types that GDB does not yet know how to read. `0xnn` is the symbol type of the misunderstood 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_syntab`, 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.

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. 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 the following documentation for more discussion of debugging targets.

- “Active Targets” on page 131
- “Commands for Managing Targets” on page 132
- “Choosing Target Byte Order” on page 134
- “Remote Debugging” on page 135
- “The GDB Remote Serial Protocol” on page 135

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 use the `gdb a.out` command, 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 a program’s read-write memory—variables and so on—plus machine status, while 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 “Commands to Specify Files” on page 125). To specify as a target a process that is already running, use the `attach` command (see “Debugging a Running Process” on page 40).

Commands for Managing Targets

The following are some commands for 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`, which designates what you use 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 use **Return** 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 “Commands to Specify Files” on page 125).

`help target name`

Describe a particular target, including any parameters (using `name` for the specific target) 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 “Commands to Specify Files” on page 125.

`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 following output.

The current BFD target is “auto”.

The following are some common targets (available, or not, depending on the GDB configuration). Different targets are available on different configurations of GDB; your configuration may have more or fewer targets.

`target exec program`

An executable file. `target exec program` is like `exec-file program`.

`target core filename`

A core dump file. `target core filename` is like `core-file filename`.

`target remote dev`

Remote serial target in GDB-specific protocol. The `dev` argument specifies what serial device to use for the connection (for example, `/dev/ttya`). See “Remote Debugging” on page 135. `target remote` now 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`

CPU simulator. See “Simulated CPU Target” on page 155.

`target udi keyword`

Remote AMD29K target, using the AMD UDI protocol. The `keyword` argument specifies which 29K board or simulator to use. See “The UDI Protocol for AMD29K” on page 145.

`target amd-eb dev speed prog`

Remote PC-resident AMD EB29K board, attached over serial lines. `dev` is the serial device, as for `target remote`; `speed` allows you to specify the linespeed; and `prog` is the name of the program to be debugged, as it appears to DOS on the PC. See “The EBMON Protocol for AMD29K” on page 145.

`target hms dev`

A Hitachi SH, H8/300, or H8/500 board, attached using a serial line to a host. Use special commands, `device` and `speed`, to control the serial line and the

communications speed used. See “GDB and Hitachi Microprocessors” on page 152.

`target nindy devicename`

An Intel 960 board controlled by a Nindy Monitor. *device-name* is the name of the serial device to use for the connection (for example, `/dev/ttya`); see “GDB with a Remote i960 (Nindy)” on page 144 for more information.

`target st2000 dev speed`

A Tandem ST2000 phone switch, running Tandem’s STD-BUG protocol. *dev* is the name of the device attached to the ST2000 serial line; *speed* is the communication line speed. The arguments are not used if GDB is configured to connect to the ST2000, using TCP or Telnet. See “GDB with a Tandem ST2000” on page 148.

`target vxworks machinename`

A VxWorks system, attached using TCP/IP. The argument, *machinename*, is the target system’s machine name or IP address. See “GDB and VxWorks” on page 148.

`target cpu32bug dev`

CPU32BUG monitor, running on a CPU32 (M68K) board.

`target op50n dev`

OP50N monitor, running on an OKI HPPA board.

`target w89k dev`

W89K monitor, running on a Winbond HPPA board.

`target est dev`

EST-300 ICE monitor, running on a CPU32 (M68K) board.

`target rom68k dev`

ROM 68K monitor, running on an IDP board.

`target array dev`

Array Tech LSI33K RAID controller board.

`target sparclite dev`

Fujitsu SPARClite boards, used only for the purpose of loading. You must use an additional command to debug the program like, for example, `target remote dev`, using GDB standard remote protocol.

Choosing Target Byte Order

To choose which byte order to use with a target system, use the `set endian big` and `set endian little` commands. Use the `set endian auto` command to instruct GDB to use the byte order associated with the executable. See the current setting for byte order with the `show endian` command.

WARNING! Currently, only embedded MIPS configurations support dynamic selection of target byte order.

Remote Debugging

If you are trying to debug a program running on a machine that cannot run GDB in the usual way, it is often useful to use remote debugging. For example, you might use remote debugging on an operating system kernel, or on a small system which does not have a general purpose operating system powerful enough to run a full-featured debugger.

Some configurations of GDB have special serial or TCP/IP interfaces to make this work with particular debugging targets. In addition, GDB comes with a generic serial protocol (specific to GDB, but not specific to any particular target system) which you can use if you write the remote stubs—the code that runs on the remote system to communicate with GDB.

Other remote targets may be available in your configuration of GDB; use `help target` to list them.

The GDB Remote Serial Protocol

The following documentation discusses the GDB remote serial protocol. You must link with your program using a few special-purpose subroutines called *stubs* that implement the GDB remote serial protocol.

- “What the Stub Can Do” on page 137
- “What You Must Do for the Stub” on page 137
- “Putting It All Together” on page 139
- “Communication Protocol” on page 140
- “Using the gdbserver Program” on page 141
- “Using the gdbserve.nlm Program” on page 143
- “GDB with a Remote i960 (Nindy)” on page 144
- “The UDI Protocol for AMD29K” on page 145
- “GDB with a Tandem ST2000” on page 148
- “GDB and VxWorks” on page 148
- “GDB and SPARClet” on page 150
- “Connecting to SPARClet” on page 151
- “SPARClet Download” on page 151
- “GDB and Hitachi Microprocessors” on page 152

- “GDB and Remote MIPS Boards” on page 153

To debug a program running on another machine (the debugging *target* machine), you must use the following directions.

1. Arrange for all the usual prerequisites for the program to run by itself. For example, for a C program, you need the following three prerequisites.
 - A startup routine to set up the C runtime environment (these usually have a name like `crt0`). The startup routine may be supplied by a hardware supplier, so you may have to write your own.
 - You probably need a C subroutine library to support your program’s subroutine calls, notably managing input and output.
 - A way of getting your program to the other machine—for example, a download program. These are often supplied by manufacturers, so you may have to write your own from hardware documentation.
2. Arrange for your program to use a serial port to communicate with the machine where GDB is running (the *host* machine). In general terms, the scheme follows a standard protocol.
 - On the *host*
GDB already understands how to use this protocol; when everything else is set up, use the `target remote` command (see “Commands for Managing Targets” on page 132).
 - On the *target*
You must link with your program using a few special-purpose subroutines that implement the GDB remote serial protocol. The file containing these subroutines is called a debugging stub.

On certain remote targets, you can use an auxiliary program, `gdbserver`, instead of linking a stub into your program. See “Using the `gdbserver` Program” on page 141 for details.

The debugging stub is specific to the architecture of the remote machine; for example, use `sparc-stub.c` to debug programs on SPARC boards. The following working remote stubs are distributed with GDB.

- `sparc-stub.c`
For SPARC architectures.
- `m68k-stub.c`
For Motorola 680x0 architectures.
- `i386-stub.c`
For Intel 386 and compatible architectures.

The `README` file in the GDB distribution may list other recently added stubs.

What the Stub Can Do

The debugging *stub* for your architecture is what supplies the following three subroutines.

`set_debug_traps`

This routine arranges for `handle_exception` to run when your program stops. You must call this subroutine explicitly near the beginning of your program.

`handle_exception`

This is the central workhorse, but your program never calls it explicitly—the setup code arranges for `handle_exception` to run when a trap is triggered.

`handle_exception` takes control when your program stops during execution (for example, on a breakpoint), and mediates communications with GDB on the host machine. This is where the communications protocol is implemented; `handle_exception` acts as the GDB representative on the target machine; it begins by sending summary information on the state of your program, then continues to execute, retrieving and transmitting any information GDB needs, until you execute a GDB command that makes your program resume; at that point, `handle_exception` returns control to your own code on the target machine.

`breakpoint`

Use this auxiliary subroutine to make your program contain a breakpoint. Depending on the particular situation, this may be the only way for GDB to get control. For instance, if your target machine has some sort of interrupt button, you won't need to call this; pressing the interrupt button transfers control to `handle_exception`; in effect, the transfer is to GDB. On some machines, simply receiving characters on the serial port may also trigger a trap; again, in that situation, you don't need to call `breakpoint` from your own program—simply running `target remote` from the host GDB session gets control.

Call `breakpoint` if none of these is true, or if you simply want to make certain your program stops at a predetermined point for the start of your debugging session.

What You Must Do for the Stub

The debugging stubs that come with GDB are set up for a particular chip architecture, having no information about the rest of the target machine being debugged.

First of all, you need to tell the stub how to communicate with the serial port with the following subroutines.

`int getDebugChar()`

Write this subroutine to read a single character from the serial port. It may be identical to `getchar` for your target system; a different name is used to allow you to distinguish the two if you wish.

```
void putDebugChar (int)
```

Write this subroutine to write a single character to the serial port. It may be identical to `putchar` for your target system; a different name is used to allow you to distinguish the two if you wish.

If you want GDB to be able to stop your program while it is running, you need to use an interrupt-driven serial driver, and arrange for it to stop when it receives a `^C` (`\003`, the **Ctrl-C** key assignment). That is the character which GDB uses to tell the remote system to stop.

Getting the debugging target to return the proper status to GDB probably requires changes to the standard stub; one quick and dirty way is to just execute a breakpoint instruction (the “dirty” part is that GDB reports a `SIGTRAP` instead of a `SIGINT`).

Other routines you need to supply are the following.

```
void exceptionHandler (int exception_number, void *exception_address)
```

Write this function to install `exception_address` in the exception handling tables.

You need to do this because the stub does not have any way of knowing what the exception handling tables on your target system are like (for example, the processor’s table might be in ROM, containing entries which point to a table in RAM). `exception_number` is the exception number which should be changed; its meaning is architecture-dependent (for example, different numbers might represent divide by zero, misaligned access, etc). When this exception occurs, control should be transferred directly to `exception_address`, and the processor state (stack, registers, and so on) should be just as it is when a processor exception occurs. So if you want to use a jump instruction to reach `exception_address`, it should be a simple jump, not a jump to subroutine.

For the 386, `exception_address` should be installed as an interrupt gate so that interrupts are masked while the handler runs. The gate should be at privilege level 0 (the most privileged level). The SPARC and 68K stubs are able to mask interrupt themselves without help from `exceptionHandler`.

```
void flush_i_cache()
```

(sparc and sparclite only) Write this subroutine to flush the instruction cache, if any, on your target machine. If there is no instruction cache, this subroutine may be a no-op.

On target machines that have instruction caches, GDB requires this function to make certain that the state of your program is stable.

You must also make sure the following library routine is available.

```
void *memset(void *, int, int)
```

This is the standard library function, `memset`, which sets an area of memory to a known value. If you have one of the free versions of `libc.a`, `memset` can be found

there; otherwise, you must either obtain it from your hardware manufacturer, or write your own.

If you do not use the GNU C compiler, you may need other standard library subroutines as well; this varies from one stub to another, but in general the stubs are likely to use any of the common library subroutines which `gcc` generates as inline code.

Putting It All Together

In summary, when your program is ready to debug, use the following steps.

1. Make sure you have the supporting low-level routines (see “What You Must Do for the Stub” on page 137): `getDebugChar`, `putDebugChar`, `flush_i_cache`, `memset`, `exceptionHandler`.
2. Insert these lines near the top of your program:

```
set_debug_traps();
breakpoint();
```

For the Motorola 680x0 stub only, you need to provide a variable called `exceptionHook`. Normally you just use `void (*exceptionHook)() = 0;`, but if before calling `set_debug_traps`, you set it to point to a function in your program, that function is called when GDB continues after stopping on a trap (for example, bus error). The function indicated by `exceptionHook` is called with one parameter: an `int` which is the exception number.

3. Compile and link together: your program, the GDB debugging stub for your target architecture, and the supporting subroutines.
4. Make sure you have a serial connection between your target machine and the GDB host, and identify the serial port on the host.
5. Download your program to your target machine (or get it there by whatever means the manufacturer provides), and start it.
6. To start remote debugging, run GDB on the host machine, and specify as an executable file the program that is running in the remote machine. This tells GDB how to find your program’s symbols and the contents of its pure text.

Then establish communication using the `target remote` command. Its argument specifies how to communicate with the target machine—either via a devicename attached to a direct serial line, or a TCP port (usually to a terminal server which in turn has a serial line to the target). For example, to use a serial line connected to the device named `/dev/ttyb`, use `target remote /dev/ttyb`.

To use a TCP connection, use an argument of the form `host:port`. For example, to connect to port 2828 on a terminal server named `manyfarms`, use the following command.

```
target remote manyfarms:2828.
```

Now you can use all the usual commands to examine and change data and to step and

continue the remote program.

To resume the remote program and stop debugging it, use the `detach` command.

Whenever GDB is waiting for the remote program, if you use the interrupt character sequence (often, **Ctrl-C**), GDB attempts to stop the program. This may or may not succeed, depending in part on the hardware and the serial drivers the remote system uses. If you type the interrupt character once again, GDB displays the following output:

```
Interrupted while waiting for the program.  
Give up (and stop debugging it)? (y or n)
```

If you use the `y` key, GDB abandons the remote debugging session. (If you decide you want to try again later, you can use `target remote` again to connect once more.) If you use the `n` key, GDB goes back to waiting.

Communication Protocol

The stub files provided with GDB implement the target side of the communication protocol, and the GDB side is implemented in the `GDB remote.c` source file.

Normally, you can simply allow these subroutines to communicate, and ignore the details. If you're implementing your own stub file, you can still ignore the details: start with one of the existing stub files. `sparc-stub.c` is the best organized, and therefore the easiest to read. However, there may be occasions when you need to know something about the protocol; for example, if there is only one serial port to your target machine, you might want your program to do something special if it recognizes a packet meant for GDB.

All GDB commands and responses (other than acknowledgments, which are single characters) are sent as a packet which includes a checksum. A packet is introduced with the `$` character, and ends with the `#` character, followed by a two-digit checksum, as in the following input example.

```
$packet info#checksum
```

checksum is computed as the modulo 256 sum of the *packet info* characters.

When either the host or the target machine receives a packet, the first response expected is an acknowledgement: a single character, either `+` (to indicate the package was received correctly) or `-` (to request retransmission). The host (using GDB) sends commands, and the target (with the debugging stub incorporated) sends data in response. The target also sends data when your program stops.

Command packets are distinguished by their first character, which identifies the kind of command. The following commands are currently supported (for a complete list of commands, look in the `gdb/remote.c` directory).

`g`

Requests the values of CPU registers.

- G**
Sets the values of CPU registers.
- maddr, count*
Read *count* bytes at location, *addr*.
- Maddr, count:...*
Write *count* bytes at location, *addr*.
- c c addr*
Resume execution at the current address (or *addr*, if supplied).
- s s addr*
Step the target program for one instruction, from either the current program counter or from *addr*, if supplied.
- k**
Kill the target program.
- ?**
Report the most recent signal. To allow you to take advantage of the GDB signal handling commands, one of the functions of the debugging stub is to report CPU traps as the corresponding POSIX signal values.
- T**
Allows the remote stub to send only the registers that GDB needs to make a quick decision about single-stepping or conditional breakpoints. This eliminates the need to fetch the entire register set for each instruction through which GDB steps. GDB then implements a write-through cache for registers and only re-reads the registers if the target has run.
- If you have trouble with the serial connection, use the `set remotedebug` command. GDB then will report on all packets sent back and forth across the serial line to the remote machine. The packet-debugging information is printed on the GDB standard output stream. `set remotedebug off` turns it off, and `show remotedebug` shows you the current state.

Using the `gdbserver` Program

`gdbserver` is a control program for UNIX-like systems, allowing you to connect your program with a remote GDB using the `target remote` command, without linking in the usual debugging stub.

`gdbserver` is not a complete replacement for the debugging stubs, because it requires essentially the same operating-system facilities that GDB itself does. In fact, a system that can run `gdbserver` to connect to a remote GDB could also run GDB locally.

`gdbserver` is sometimes useful nevertheless, because it is a much smaller program than GDB itself. It is also easier to port than all of GDB, so you may be able to get

started more quickly on a new system by using `gdbserver`. Finally, if you develop code for real-time systems, you may find that the tradeoffs involved in real-time operation make it more convenient to do as much development work as possible on another system, for example by cross-compiling. You can use `gdbserver` to make a similar choice for debugging.

GDB and `gdbserver` communicate using either a serial line or a TCP connection, using the standard GDB remote serial protocol. The following discussions detail the connections of the target machine and the host machine.

On the target machine

You need to have a copy of the program you want to debug. `gdbserver` does not need your program's symbol table, so you can strip the program if necessary to save space. GDB on the host system does all the symbol handling. To use the server, you must tell it how to communicate with GDB; the name of your program; and the arguments for your program. The syntax is:

```
target> gdbserver comm program [args...].
```

`comm` is either a device name (to use a serial line) or a TCP hostname and portnumber. For example, to debug Emacs with the argument, `foo.txt`, and communicate with GDB over the serial port, `/dev/com1`, use the following.

```
target> gdbserver /dev/com1 emacs foo.txt.
```

`gdbserver` waits passively for the host GDB to communicate with it. To use a TCP connection instead of a serial line, use the following.

```
target> gdbserver host:2345 emacs foo.txt.
```

The only difference from the previous example is the first argument, specifying that you are communicating with the host GDB with TCP. The `host:2345` argument means that `gdbserver` is to expect a TCP connection from a host machine to local TCP port 2345; the `host` part is ignored. You can choose any number you want for the port number as long as it does not conflict with any TCP ports already in use on the target system (for example, 23 is reserved for `telnet`).*

You must use the same port number with the host GDB `target remote` command.

On the GDB host machine

You need an unstripped copy of your program, since GDB needs symbols and debugging information.

Start up GDB as usual, using the name of the local copy of your program as the first argument. You may also need the `--baud` option if the serial line is running at anything other than 9600 bps.

After that, use `target remote` to establish communications with `gdbserver`.

Its argument is either a device name (usually a serial device like `/dev/ttyb`) or a

* If you choose a port number that conflicts with another service, `gdbserver` prints an error message and exits.

TCP port descriptor in the form, *host:port*. For example, the input,
`target remote /dev/ttyb`, communicates with the server via serial line,
 designated with the pathname, `/dev/ttyb`.

`target remote the-target:2345` communicates with a TCP connection to port
 2345 on host, *the-target*. For TCP connections, you must start up `gdbserver`
 prior to using the `target remote` command. Otherwise you may get an error
 whose text depends on the host system, but which usually looks something like
Connection refused in the declaration.

Using the `gdbserve.nlm` Program

`gdbserve.nlm` is a control program for NetWare systems, allowing you to connect
 your program with a remote GDB `target remote` command.

GDB and `gdbserve.nlm` communicate using a serial line, with the standard GDB
 remote serial protocol. The following discussions detail the connections of the target
 machine and the host machine.

On the target machine

You need to have a copy of the program you want to debug. `gdbserve.nlm` does
 not need your program's symbol table, so you can strip the program if necessary
 to save space. GDB on the host system does all the symbol handling. To use the
 server, you must tell it: how to communicate with GDB, the name of your
 program, and the arguments for your program. The syntax is the following.

```
load gdbserve [ BOARD=board ] [ PORT=port ]
               [ BAUD=baud ] program [ args ... ]
```

board and *port* specify the serial line; *baud* specifies the baud rate used by the
 connection. *port* and *node* default to 0, *baud* defaults to 9600 bps. For example,
 to debug Emacs with the argument, `foo.txt`, in order to communicate with GDB
 over serial port number 2 or board 1 using a 19200 bps connection, use the
 following declaration.

```
load gdbserve BOARD=1 PORT=2 BAUD=19200 emacs foo.txt
```

On the GDB host machine, you need an unstripped copy of your program, since
 GDB needs symbols and debugging information. Start up GDB as usual, using the
 name of the local copy of your program as the first argument. (You may also need
 the `--baud` option if the serial line is running at anything other than 9600 bps.

After that, use `target remote` to establish communications with `gdbserve.nlm`.
 Its argument is a device name (usually a serial device, like `/dev/ttyb`). For
 example, `(gdb) target remote /dev/ttyb` communicates with the server via
 serial line, `/dev/ttyb`.

GDB with a Remote i960 (Nindy)

Nindy is a ROM monitor program for Intel 960 target systems. When GDB is configured to control a remote Intel 960 using *Nindy*, you can tell GDB how to connect to the 960 in the following ways.

- Through command line options specifying serial port, version of the *Nindy* protocol, and communications speed;
- By responding to a prompt on startup;
- By using the `target` command at any point during your GDB session. See “Commands for managing targets” on page Commands for Managing Targets.

Startup with Nindy

If you start GDB without using any command-line options, you are prompted for what serial port to use, *before* you reach the ordinary GDB prompt:

```
attach /dev/ttyNN -- specify NN, or "quit" to quit:
```

Respond to the prompt with whatever suffix (after `/dev/tty`) to identify the serial port that you want to use. You can, if you choose, simply start up with no *Nindy* connection by responding to the prompt with an empty line. If you do this and later wish to attach to *Nindy*, use `target` (see “Commands for Managing Targets” on page 132).

Nindy Reset Command

`reset`

For a *Nindy* target, this command sends a “break” to the remote target system; this is only useful if the target has been equipped with a circuit to perform a hard reset (or some other interesting action) when a break is detected.

Options for Nindy

The following are the startup options for beginning your GDB session with a *Nindy-960* board attached.

`-r port`

Specify the serial port name of a serial interface to be used to connect to the target system. This option is only available when GDB is configured for the Intel 960 target architecture. You may specify `port` as any of: a full pathname (for example, `-r /dev/ttya`), a device name in `/dev` (for example, `-r ttya`), or simply the unique suffix for a specific tty (for example, `-r a`).

`-O`

An uppercase letter O, not a zero, for specifying that GDB should use the old *Nindy* monitor protocol to connect to the target system. This option is only available when GDB is configured for the Intel 960 target architecture.

WARNING! If you specify `-o`, but are actually trying to connect to a target system that expects the newer protocol, the connection fails, appearing to be a speed mismatch. GDB repeatedly attempts to reconnect at several different line speeds. You can abort this process with an interrupt.

`-brk`

Specify that GDB should first send a `BREAK` signal to the target system, in an attempt to reset it, before connecting to a Nindy target.

WARNING! Many target systems do not have the hardware that this requires; it only works with a few boards.

The standard `-b` option controls the line speed used on the serial port.

The UDI Protocol for AMD29K

GDB supports AMD's UDI (*Universal Debugger Interface*) protocol for debugging the A29K processor family. To use this configuration with AMD targets running the MiniMON monitor, you need the program, `MONTIP`, available from AMD at no charge. You can also use GDB with the UDI-conformant A29K simulator program, `ISSTIP`, also available from AMD.

`target udi keyword`

Select the UDI interface to a remote 29K board or simulator, where *keyword* is an entry in the AMD configuration file `udi_soc`. This file contains keyword entries which specify parameters used to connect to A29K targets. If the `udi_soc` file is not in your working directory, you must set the `UDICONF` environment variable to its pathname.

The EBMON Protocol for AMD29K

AMD distributes a 29K development board meant to fit in a PC, together with a DOS-hosted monitor program called `EBMON`. As a shorthand term, we use “EB29K” as a name for this development system.

To use GDB from a UNIX system to run programs on the EB29K board, connect a serial cable between the PC (which hosts the EB29K board) and a serial port on the UNIX system. In the following, we assume you've hooked the cable between the PC's `COM1` port and `/dev/ttya` on the UNIX system.

The next step is to set up the PC's port, using something like the following in DOS on the PC:

```
C:\> MODE com1:9600,n,8,1,none
```

This example—run on an MS DOS 4.0 system—sets the PC port to 9600 bps, no parity, eight data bits, one stop bit, and no “retry” action; you must match the communications parameters when establishing the UNIX end of the connection as

well.

To give control of the PC to the UNIX side of the serial line, at the `C:\>` prompt, type: `CTTY com1`. (Later, if you wish to return control to the DOS console, you can use the command `CTTY con`—but you must send it over the device that had control, in the example, over the `COM1` serial line). From the UNIX host, use a communications program such as `tip` or `cu` to communicate with the PC; for example, use `cu -s 9600 -l /dev/ttya` as input; these `cu` options specify, respectively, the linespeed and the serial port to use. If you use `tip` instead, your input would be something like `tip -9600 /dev/ttya` as input; your system may require a different name than `/dev/ttya` as the argument to `tip`. The communications parameters, including which port to use, are associated with the `tip` argument in the remote descriptions file; normally they are in the system table, `/etc/remote`.

Using the `tip` or `cu` connection, change the DOS working directory to the directory containing a copy of your 29K program, then start the PC program, `EBMON` (an EB29K control program supplied with your board by AMD).

You should see an initial display from `EBMON` similar to the one that follows, ending with the `EBMON` prompt, `#`.

Example 1: PC program, `EBMON` (an EB29K control program)

```
C:\> G:

G:\> CD \usr\joe\work29k
G:\USR\JOE\WORK29K> EBMON Am29000 PC Coprocessor Board Monitor,
version 3.0-18 Copyright 1990 Advanced Micro Devices, Inc. Written by
Gibbons and Associates, Inc.

Enter '?' or 'H' for help

PC Coprocessor Type      = EB29K
I/O Base                 = 0x208
Memory Base              = 0xd0000
Data Memory Size         = 2048KB
Available I-RAM Range    = 0x8000 to 0x1ffffff
Available D-RAM Range    = 0x80002000 to 0x801ffffff

PageSize                 = 0x400
Register Stack Size      = 0x800
Memory Stack Size        = 0x1800

CPU PRL                  = 0x3
Am29027 Available        = No
Byte Write Available     = Yes
```

```
# ~ .
```

Then exit the `cu` or `tip` program (the previous example shows the use of ‘~.’ as input at the `EBMON` prompt, #). `EBMON` keeps running, ready for GDB to resume its processing. For this example, we’ve assumed what is probably the most convenient way to make sure the same 29K program is on both the PC and the UNIX system: a PC/NFS connection that establishes the `G:` drive on the PC as a file system on the UNIX host. If you do not have PC/NFS or something similar connecting the two systems, you must arrange some other way—perhaps floppy-disk transfer—getting the 29K program from the UNIX system to the PC; GDB does not download it over the serial line.

Finally, `cd` to the directory containing an image of your 29K program on the UNIX system, and start GDB—specifying as argument the name of your 29K program, as in the following example.

```
cd /usr/joe/work29k
gdb myfoo
```

Now, use the `target` command, as in the following declaration.

```
target amd-eb /dev/ttya 9600 MYFOO
```

The previous example has the program in a file called `MYFOO`.

IMPORTANT! The filename given as the last argument to `target amd-eb` should be the name of the program as it appears to DOS. In the previous example, this is simply `MYFOO`, but in general it can include a DOS path, and, depending on your transfer mechanism, may not resemble the name on the UNIX side. At this point, you can set any breakpoints you wish; when you are ready to see your program run on the 29K board, use the GDB command, `run`.

To stop debugging the remote program, use the GDB `detach` command. To return control of the PC to its console, use `tip` or `cu` once again, after your GDB session has concluded, to attach to `EBMON`. You can then type the command `q` to shut down `EBMON`, returning control to the DOS command-line interpreter. Type `CTTY con` to return command input to the main DOS console, and type `~.` to leave `tip` or `cu`. See Example 1: “PC program, `EBMON` (an EB29K control program)” on page 146.

Remote Log

The `target amd-eb` command creates a `eb.log` file in the current working directory, to help debug problems with the connection. `eb.log` records all the output from `EBMON`, including echoes of the commands sent to it. Running `tail -f` on this file in another window often helps to understand trouble with `EBMON`, or unexpected events on the PC side of the connection.

GDB with a Tandem ST2000

To connect your ST2000 to the host system, see the manufacturer's manual. Once ST2000 is physically attached, you can run `target st2000 dev speed` to establish it as your debugging environment.

`dev` is normally the name of a serial device, such as `/dev/ttya`, connected to the ST2000 via a serial line. You can instead specify the device as a TCP connection (for example, to a serial line attached via a terminal concentrator) using the syntax, `hostname:portnumber`, where `hostname` signifies, for instance, the ST2000, the host system, and `portnumber` is the actual serial port to specify.

The `load` and `attach` commands are not defined for this target; you must load your program into the ST2000 as you normally would for standalone operation. GDB reads debugging information (such as symbols) from a separate, debugging version of the program available on your host computer.

The following auxiliary GDB commands are available to help you with the ST2000 environment:

`st2000 command`

Send a `command` to the STDEBUG monitor. See the manufacturer's manual for available commands.

`connect`

Connect the controlling terminal to the STDEBUG command monitor. When you are done interacting with STDEBUG, typing either of two keystroke sequences gets you back to the GDB command prompt: using the **Return** key, then the tilde key (~), and then the period (.) key; or the **Return** key, the tilde key, and then, simultaneously, the **Control** and uppercase **D** keys).

GDB and VxWorks

GDB enables developers to spawn and debug tasks running on networked VxWorks targets from a UNIX host. Already-running tasks spawned from the VxWorks shell can also be debugged. GDB uses code that runs on both the UNIX host and on the VxWorks target. The `gdb` program is installed and executed on the UNIX host. (It may be installed with the name, `vxgdb`, to distinguish it from GDB for debugging programs on the host itself.)

`VxWorks-timeout args`

All VxWorks-based targets now support the option `vxworks-timeout`. This option is set by the user, and `args` represents the number of seconds GDB waits for responses to `rpc`'s. You might use this if your VxWorks target is a slow software simulator or is on the far side of a thin network line.

The following information on connecting to VxWorks was current when this documentation was produced; newer releases of VxWorks may use revised procedures. To use GDB with VxWorks, you must rebuild your VxWorks kernel to

include the remote debugging interface routines in the VxWorks `rdb.a` library. To do this, define `INCLUDE_RDB` in the VxWorks `configAll.h` configuration file and rebuild your VxWorks kernel. The resulting kernel contains `rdb.a`, and spawns the source debugging task, `trdbTask`, when VxWorks is booted. For more information on configuring and remaking VxWorks, see the manufacturer's manual. Once you have included `rdb.a` in your VxWorks system image and set your UNIX execution search path to find GDB, you are ready to run GDB. From your UNIX host, run `gdb` (or `vxgdb`, depending on your installation). GDB comes up showing the prompt, `(vxgdb)`.

Connecting to VxWorks

The GDB command target lets you connect to a VxWorks target on the network. To connect to a target whose host name is `tt`, use something like the following example's declaration.

```
(vxgdb) target vxworks tt
```

GDB then displays messages like the following declarations.

```
Attaching remote machine across net...
Connected to tt.
```

GDB then attempts to read the symbol tables of any object modules loaded into the VxWorks target since it was last booted. GDB locates these files by searching the directories listed in the command search path (see "Your Program's Environment" on page 38); if it fails to find an object file, the following message displays.

```
prog.o: No such file or directory.
```

When this happens, add the appropriate directory to the search path with the GDB command path, and execute the target command again.

VxWorks Download

If you have connected to the VxWorks target and you want to debug an object that has not yet been loaded, you can use the GDB `load` command to download a file from UNIX to VxWorks incrementally. The object file given as an argument to the `load` command is actually opened twice: first by the VxWorks target in order to download the code, then by GDB in order to read the symbol table. This can lead to problems if the current working directories on the two systems differ. If both systems have had NFS mount the same filesystems, you can avoid these problems by using absolute paths. Otherwise, it is simplest to set the working directory on both systems to the directory in which the object file resides, and then to reference the file by its name, without any path. For instance, a `prog.o` program may reside in `vxpath/vw/demo/rdb` in VxWorks and in `hostpath/vw/demo/rdb` on the host. To load this program on VxWorks, use `-> cd "vxpath/vw/demo/rdb"` as input. Then, in GDB, type the following commands at the `(vxgdb)` prompt:

```
cd hostpath/vw/demo/rdb
load prog.o
```

GDB displays a response similar to the following output.

```
Reading symbol data from wherever/vw/demo/rdb/prog.o
...
Done.
```

You can also use the `load` command to reload an object module after editing and recompiling the corresponding source file.

IMPORTANT! This input makes GDB delete all currently-defined breakpoints, auto-displays, and convenience variables, and clears the value history. This is necessary in order to preserve the integrity of debugger data structures that reference the target system's symbol table.

Running Tasks with VxWorks

You can also attach to an existing task using the `attach` command as follows.

```
attach task
```

`task` is the VxWorks hexadecimal task ID. The task can be running or suspended when you attach to it. Running tasks are suspended at the time of attachment.

GDB and SPARClet

GDB enables developers to debug tasks running on SPARClet targets from a UNIX host. GDB uses code that runs on both the UNIX host and on the SPARClet target. The program, `gdb`, is installed and executed on the UNIX host.

```
timeout args
```

GDB now supports the option, `remotetimeout`. This option is set by the user; `args` represents the number of seconds GDB waits for responses.

When compiling for debugging, include the option, `-g`, to get debug information and the option, `-Ttext`, to relocate the program to where you wish to load it on the target. You may also want to add the option, `-n`, or the option, `-N`, in order to reduce the size of the sections. Use the following command input as an example (where `prog` signifies the program that you designate).

```
sparclet-aout-gcc prog.c -Ttext 0x12010000 -g -o prog -N
```

You can use `objdump` to verify that the addresses are what you intended.

```
sparclet-aout-objdump --headers --syms prog
```

Once you have set your UNIX execution search path to find GDB, you are ready to run GDB. From your UNIX host, run GDB, using the input, `gdb` (or the input specific to your host system). GDB then shows its prompt, (`gdb$let`).

Setting file to Debug

The GDB command, `file`, lets you choose which program to debug as the following example shows. GDB then attempts to read the symbol table of `prog`, the program that you designate at the prompt, (`gdb$let`).

```
file prog
```

GDB locates the file by searching the directories listed in the command search path. If the file was compiled with debug information (using the option, `-g`), source files will be searched as well. GDB locates the source files by searching the directories listed in the directory search path (see “Your Program’s Environment” on page 38). If it fails to find a file, it displays a message such as: `prog: No such file or directory`.

When this happens, add the appropriate directories to the search paths with the GDB commands, `path` and `dir`, and execute the `target` command again.

Connecting to SPARCllet

The GDB command, `target`, lets you connect to a SPARCllet target. To connect to a target on serial port called `ttya`, use the following command at the SPARCllet GDB prompt, `gdb$let`.

```
target sparcllet /dev/ttya
```

GDB displays messages like the following output.

```
Remote target sparcllet connected to /dev/ttya
main () at ../prog.c:3
Connected to ttya.
```

SPARCllet Download

Once connected to the SPARCllet target, you can use the GDB `load` command to download the file from the host to the target. The file name and load offset should be given as arguments to the `load` command. Since the file format is `a.out`, the program must be loaded to the starting address. You can use the binary utility, `objdump`, to find out what this value is. The load offset is an offset which is added to the `vma` (virtual memory address) of each of the file’s sections. For instance, if the program, `prog`, was linked to text address, `0x1201000`, with data at `0x12010160` and `bss` at `0x12010170`, in GDB, use the command, `load prog 0x12010000`, at the prompt, (`gdb$let`).

You’ll then see the following output.

```
Loading section .text, size 0xdb0 vma 0x12010000
```

If the code is loaded at a different address than that to which the program was linked, you may need to use the `section` and `add-symbol-file` commands to tell GDB where to map the symbol table.

Running and Debugging with SPARCllet

Now begin debugging the task using any of GDB’s commands: `b` (or `breakpoint`), `step`, `run`, and so on (for help with GDB commands, use the command, `help`). The following example shows what you’d do and see for execution control.

```
b main
Breakpoint 1 at 0x12010000: file prog.c, line 3.
```

The previous instruction sets a breakpoint at line 3 for the file. Then you use the command, `run`. The following is an example of what you’d then see.

```
run
```

The following is an example of the output from GDB you'd then see.

```
Starting program: prog
Breakpoint 1, main (argc=1, argv=0xeffff21c) at prog.c:3
3         char *symarg = 0;
```

Then, at your prompt, use the command, `step`, and set the next breakpoint at 4. The following is an example of what you'd then see.

```
step
4         char *execarg = "hello!";
```

GDB and Hitachi Microprocessors

GDB needs to know the following things to talk to your Hitachi SH, H8/300, or H8/500.

- That you want to use `target hms`, the remote debugging interface for Hitachi microprocessors, or `target e7000`, the in-circuit emulator for the Hitachi SH and the Hitachi 300Hh. (`target hms` is the default when GDB is configured specifically for the Hitachi SH, H8/300, or H8/500.)
- What serial device connects your host to your Hitachi board (the first serial device available on your host is the default).
- What speed to use over the serial device.

Connecting to Hitachi Boards

Use the special GDB `device port` command if you need to explicitly set the serial device; `port` is the first available port on your host. This is only necessary on UNIX hosts, where it is typically something like `/dev/ttya`.

GDB has another special command to set the communications speed. `speed bps` is only used from UNIX hosts. On DOS hosts, set the line speed as usual from outside GDB with the DOS `mode` command; for instance, `mode com2:9600,n,8,1,p` sets a 9600 bps connection).

The `device` and `speed` commands are available only when you use a UNIX host to debug your Hitachi microprocessor programs. If you use a DOS host, GDB depends on an auxiliary terminate-and-stay-resident program called `asynctsr` to communicate with the development board through a PC serial port. You must also use the DOS `mode` command to set up the serial port on the DOS side.

Using the E7000 In-circuit Emulator

You can use the e7000 in-circuit emulator to develop code for either the Hitachi SH or the H8/300H. Use one of the following forms of the `target e7000` command to connect GDB to your H7000.

```
target e7000 port speed
```

Use this form if your e7000 is connected to a serial port. The *port* argument identifies what serial port to use (for example, *com2*). The third argument is the line speed in bits per second (for example, 9600).

```
target e7000 hostname
```

If your e7000 is installed as a host on a TCP/IP network, you can just specify its hostname; GDB uses *telnet* to connect.

Special GDB Commands for Hitachi Micros

Some GDB commands are available only on the H8/300 or the H8/500 configurations:

```
set machine h8300
```

```
set machine h8300h
```

Condition GDB for one of the two variants of the H8/300 architecture with *set machine*. You can use *show machine* to check which variant is currently in effect.

```
set memory mod
```

```
show memory
```

Specify which H8/500 memory model (*mod*) you are using with *set memory*; check which memory model is in effect with *show memory*. The accepted values for *mod* are *small*, *big*, *medium*, and *compact*.

GDB and Remote MIPS Boards

GDB can use the MIPS remote debugging protocol to talk to a MIPS board attached to a serial line, configuring GDB with *--target-mips-idt-ecoff*.

Use the following GDB commands to specify the connection to your target board.

```
target mips port
```

To run a program on the board, start up GDB with the name of your program as the argument. To connect to the board, use the command *target mips port*, where *port* is the name of the serial port connected to the board. If the program has not already been downloaded to the board, you may use the *load* command to download it. You can then use all the usual GDB commands.

For example, the following sequence connects to the target board through a serial port, and loads and runs a program, *prog*, called through the debugger.

```
host$ gdb prog
GDB is free software ...
target mips /dev/ttyb
load prog
run
```

```
target mips hostname:portnumber
```

On some GDB host configurations, you can specify a TCP connection (for instance, to a serial line managed by a terminal concentrator) instead of a serial port, using *hostname:portnumber* syntax.

GDB also supports the following special commands for MIPS targets.

```
set processor args
show processor
```

Use the `set processor` command to set the type of MIPS processor when you want to access processor-type-specific registers. For example, the input, `set processor r3041`, tells GDB to use the CPO registers appropriate for the 3041 chip. Use the `show processor` command to see what MIPS processor GDB is using. Use the `info reg` command to see what registers GDB is using.

```
set mipsfpu double
set mipsfpu single
set mipsfpu none
show mipsfpu
```

If your target board does not support the MIPS floating point coprocessor, you should use the command `set mipsfpu none` (if you need this, you may wish to put the command in your `.gdbinit` file). This tells GDB how to find the return value of functions which return floating point values. It also allows GDB to avoid saving the floating point registers when calling functions on the board. If you are using a floating point coprocessor with only single precision floating point support, as on the R4650 processor, use the command `set mipsfpu single`. The default double precision floating point coprocessor may be selected using `set mipsfpu double`.

In previous versions the only choices were double precision or no floating point, so `set mipsfpu on` will select double precision and `set mipsfpu off` will select no floating point. As usual, you can inquire about the `mipsfpu` variable with `show mipsfpu`.

```
set remotedebug n
show remotedebug
```

You can see some debugging information about communications with the board by setting the `remotedebug` variable. If you set it to 1 using `set remotedebug 1`, every packet is displayed. If you set it to 2, every character is displayed. You can check the current value at any time with the command, `show remotedebug`.

```
set timeout seconds
set retransmit-timeout seconds
show timeout
show retransmit-timeout
```

You can control the timeout used while waiting for a packet, in the MIPS remote protocol, with the `set timeout seconds` command. The default is 5 seconds. Similarly, you can control the timeout used while waiting for an acknowledgement of a packet with the `set retransmit-timeout seconds` command. The default is 3 seconds. You can inspect both values with `show timeout` and `show retransmit-timeout`.

IMPORTANT! These commands are available *only* when GDB is configured for a `--target-mips-idt-ecoff` target.

The timeout set by `set timeout` does not apply when GDB is waiting for your program to stop. In that case, GDB waits forever because it has no way of knowing how long the program is going to run before stopping.

Simulated CPU Target

For some configurations, GDB includes a CPU simulator that you can use instead of a hardware CPU to debug your programs. Currently, a simulator is available when GDB is configured to debug Zilog Z8000 or Hitachi microprocessor targets. For the Z8000 family, `target sim` simulates either the Z8002 (the unsegmented variant of the Z8000 architecture) or the Z8001 (the segmented variant). The simulator recognizes which architecture is appropriate by inspecting the object code.

`target sim`

Debug programs on a simulated CPU (the specific CPU depending on the GDB configuration).

After specifying this target, you can debug programs for the simulated CPU in the same style as programs for your host computer; use the `file` command to load a new program image, the `run` command to run your program, and so on.

As well as making available all the usual machine registers (see “Registers” on page 91 for information about `info reg`), this debugging target provides three additional items of information as specially named registers:

- `cycles`
Counts clock-ticks in the simulator.
- `insts`
Counts instructions run in the simulator.
- `time`
Execution time in 60ths of a second.

You can refer to these values in GDB expressions with the usual conventions; for example, `b fputc if $cycles>5000` sets a conditional breakpoint that suspends only after at least 5000 simulated clock ticks.

Controlling GDB

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, see “Print Settings” on page 85; other settings are described in the following documentation.

- “Prompt” on page 158
- “Command Editing” on page 158
- “Command History” on page 158
- “Screen Size” on page 160
- “Numbers” on page 160
- “Optional Warnings and Messages” on page 161

Prompt

GDB indicates its readiness to read a command by printing a string, normally called the (`gdb`) *prompt*. You can change the prompt string with the `set prompt` command. For instance, when debugging with GDB, it is useful to change the prompt in one of the GDB sessions so that you can always tell to which one you are talking.

IMPORTANT! `set prompt` no longer adds a space for you after the prompt you set. This allows you to set a prompt that ends in a space or one that does not end in a space.

```
set prompt newprompt
```

Directs GDB to use *newprompt* as its prompt string henceforth.

```
show prompt
```

Prints a line such as `gdb's prompt is: %` for you to view.

Command Editing

GDB reads its input commands using the *readline* interface. This GNU library provides consistent behavior for programs with 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.

Command History

GDB can keep track of the commands you type during your debugging sessions, so that you can be certain of precisely what happened. Use the following commands to manage the GDB command history facility.

```
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 in the following. This file defaults to the value of the environment variable `GDBHISTFILE`, or to `./.gdb_history` 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 the following.

```
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.

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. Use the **Return** key when you want to continue the output, or type `q` (a shortcut for `quit`) or `o` 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 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.

Numbers

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 the commandline input of `set radix 012`, `set radix 10.`, or `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.

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 announcing that the symbol table for a source file is being read; see `symbol-file` in “Commands to Specify Files” on page 125.

```
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 “Errors Reading Symbol Files” on page 129).

```
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 what sometimes seems to be a lot of stupid questions to confirm certain commands. For example, if you try to run a program which is already running, after having already input the `run` command, you will see something like the following onscreen.

```
run
```

```
The program being debugged has been started already.
```

```
Start it from the beginning? (y or n)
```

If you are willing to face the consequences of your own commands, you can disable

this “feature” with the following commands.

```
set confirm off
```

Disables confirmation requests.

```
set confirm on
```

Enables confirmation requests (the default).

```
show confirm
```

Displays state of confirmation requests.

16

Canned Sequences of Commands

Aside from breakpoint commands (see “Breakpoint Command Lists” on page 56), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

The following documentation provides these discussions for this subject.

- “User-defined Commands” (below)
- “User-defined Command Hooks” on page 165
- “Command Files” on page 165
- “Commands for Controlled Output” on page 166

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 with `$arg0 ...$arg9`.

The following example shows the usage of a user-defined command is a sequence.

```
define adder
  print $arg0 + $arg1 + $arg2
```

To execute the command, a `adder 1 2 3` command declaration shows the definition of

the command, `adder`, printing the sum of its three arguments.

IMPORTANT! The arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior functions calls.

`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). There can then optionally be a line `else`, followed by a series of commands that are only executed if the expression was false. The end of the list is marked by a line containing `end`.

`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 command.

`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.

User-defined Command Hooks

You may define *hooks*, which are a special kind of user-defined command. Whenever you run a `foo` command, if `hook-foo` is the defined command, it is executed (with no arguments) before that command. In addition, a `stop` pseudo-command 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, but treat them normally during normal execution, you could define the following debugging input.

```
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, but not for command aliases; you should define a hook for the basic command name, such as `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 used had a chance to run).

If you try to define a hook which does not match any known command, you get a warning from the `define` command.

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 automatically executes commands from its *init files* (named `.gdbinit`). GDB reads the init file (if any) in your home directory, then processes command line options and operands, and then reads the init file (if any) in the current working directory. This is so the init file in your home directory can set

options (such as `set complaints`) which affect the processing of the command line options and operands. The init files are not executed if you use the `-nx` option; see “Choosing Modes” on page 26. On some configurations of GDB, the init file is known by a different name (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.

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.

The following documentation describes three commands that are useful for generating exactly the output that you want.

```
echo text
```

Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as `\n` to print a newline.

IMPORTANT! 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 a `foo -` fragment statement, use `echo \ and foo - \` as a command with a backslash at the end of the declaration. As in C, this command continues the declaration onto subsequent lines.

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

Consider the following example.

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

The previous example shows output that produces the same output as the following declaration.

```
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 “Expressions” on page 78 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 “Output Formats” on page 81 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, as in the following example.

```
printf (string, expressions...);
```

For example, you can print two values in hex like the following declaration.

```
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.

Insight, the GNUPro Debugger GUI

Copyright © 1991-2000 Red Hat.

GNUPro[®], the GNUPro logo, the Cygnus logo, Insight[™], Cygwin[™], eCos[™] and Source-Navigator[™] are all trademarks of Red Hat.

All other brand and product names, trademarks and copyrights are the property of their respective owners.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation.

For licenses and use information, see *Getting Started Guide*.

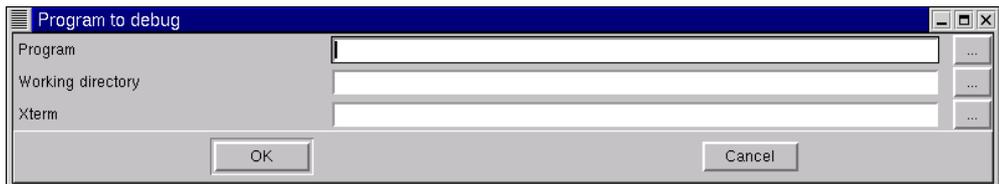
1

Insight, GDB's Alternative Interface

The following documentation serves as a general reference for GNUPro Toolkit's graphical user interface, its visual debugger, Insight; for more information, see also Insight's **Help** menu for discussion of general functionality and use of menus, buttons or other features and "Examples of Debugging with Insight" on page 199 for working with Insight.

1. From Source-Navigator, select **Tools** → **Debugger**. The **Program to debug** window displays.

Figure 1: Program to debug window



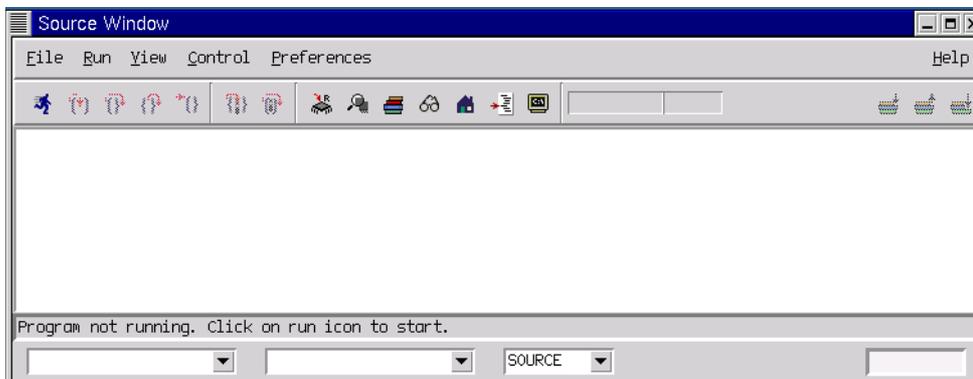
2. Click **OK**. Insight launches, displaying the **Source Window** (Figure 2). For a native project, click the **Run** button. For an embedded project, click **Run** and then click the **Continue** button. For more information on Insight, see its **Help** menu.

WARNING! Having an inactive debugging session open when starting another debugging session with GNUPro Toolkit will close all projects. All work will be unrecoverable.

Using the Source Window

When Insight first launches, it displays an empty **Source Window** (Figure 2).

Figure 2: Source Window



The menu selections in the **Source Window** are **File**, **Run**, **View**, **Control**, **Preferences** and **Help**. See “Source Window Menus and Display Features” on page 179 for more descriptions of the **Source Window**. To work with the other windows for debugging purposes specific to your project, use the **View** menu or the buttons in the toolbar (Figure 7) and see the following documentation.

- “Using the Stack Window” on page 182
- “Using the Registers Window” on page 183
- “Using the Memory Window” on page 184
- “Using the Watch Expressions Window” on page 186
- “Using the Local Variables Window” on page 188
- “Using the Breakpoints Window” on page 191
- “Using the Console Window” on page 194
- “Using the Function Browser Window” on page 195
- “Using the Processes Window for Threads” on page 197
- “Using the Help Window” on page 198

To open a specific file as a project for debugging, select **File** → **Open** in the **Source Window**. The file’s contents will then be passed to the GDB interpreter for execution. To start debugging, click the **Run** button (Figure 3) from the **Source Window**.

Figure 3: Run button



When the debugger runs, the button turns into the **Stop** button (Figure 4).

Figure 4: Stop button

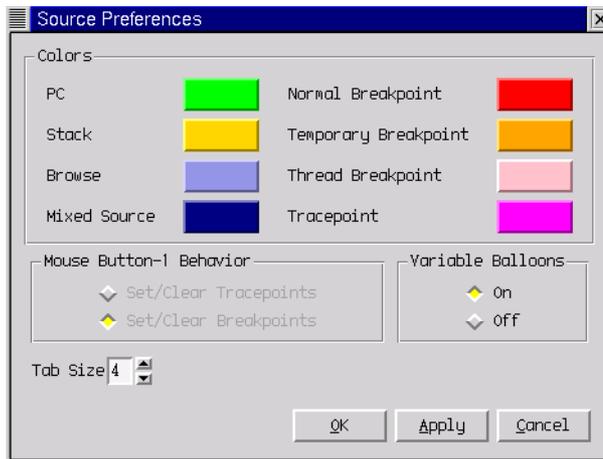


The **Stop** button interrupts the debugging process for a project, *provided that* the underlying hardware and protocols *support* such interruptions. Generally, machines that are connected to boards cannot interrupt programs on those boards, so the **Stop** button has no functionality (it will appear unavailable, or “grayed out”). For more information on the toolbar buttons, see Figure 7.

WARNING! When debugging a target, do not click on the **Run** button during an active debugging process, or it will de-activate the process. The **Run** button will become the **Stop** button and Insight will lose connection with the target.

To specify preferences of how source code appears and to change debugging settings, select **Preferences** → **Source** from the **Source Window**. The **Source Preferences** dialog opens (Figure 5).

Figure 5: Source Preferences dialog



Left-click any of the colored squares to open the **Choose color** dialog, with which you modify the display colors of the **Source Window**.

Mouse Button-1 Behavior sets and clears either breakpoints or tracepoints (points in the source code, with an associated text string); the default is for setting breakpoints.

Variable Balloons lets you display a balloon of text whenever the cursor is over a variable in the **Source Window**; the balloon displays the value of the variable (see Figure 11 on page 178 for an example). **On** is the default selection.

Selecting **Tab Size** sets the number of spaces for a tab character in the **Source Window**.

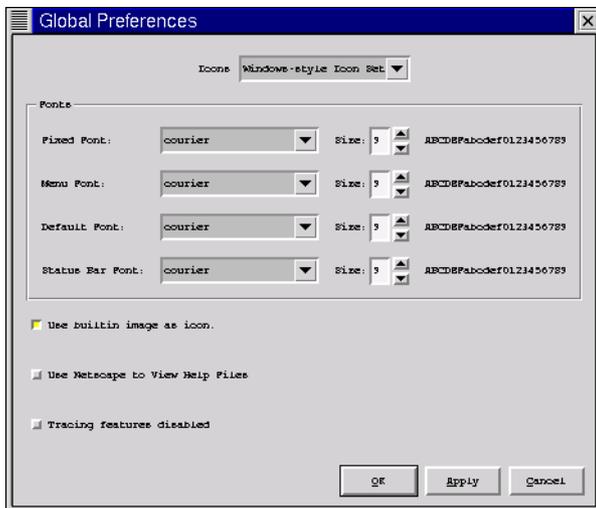
The **Source Window** has the following functionality and display features when using the **Source Preferences** dialog settings.

- When the executable is running in a debugging process, the location of the current program counter displays as a line with a colored background (**PC**).
- When the executable has finished running, the background color changes (**Browse**).
- When looking at a stack backtrace, the background color changes to another

different color (**Stack**).

To set other preferences for a debugging session, select **Preferences** → **Global** from the **Source Window**. The **Global Preferences** dialog opens (Figure 6) where you select a specific font and type size for the text in the windows for Insight.

Figure 6: Global Preferences dialog



Icons allows you to select the appearance of the toolbar buttons as the **Windows-style Icon Set** (the default; see Figure 7) or the **Basic Icon Set** (see Insight's **Help** menu for more information).

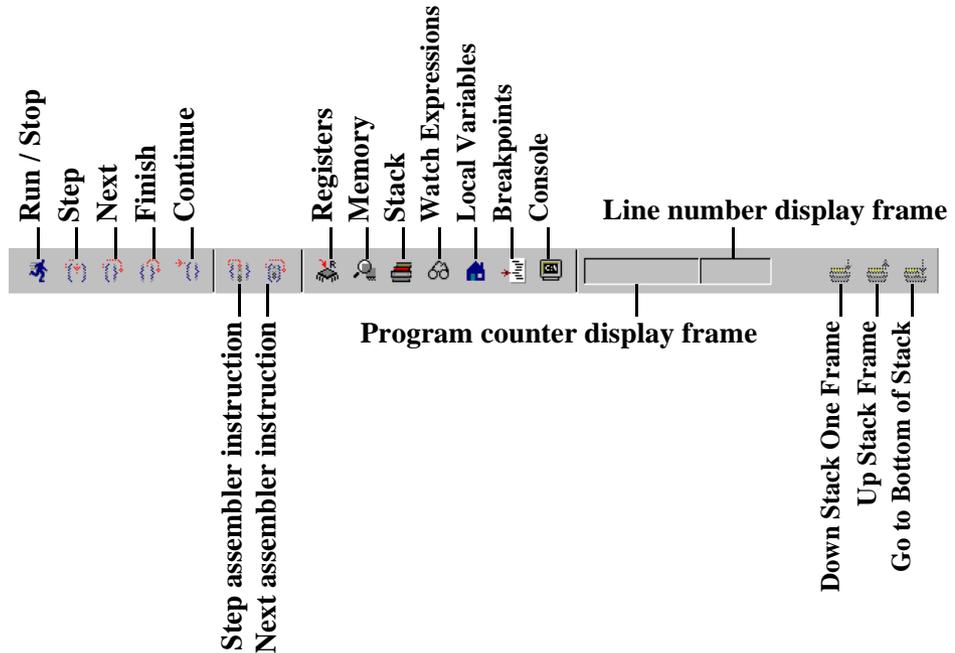
Fonts is for selecting font family and size.

- **Fixed Font** sets the font for the source code display panes.
- **Default Font** sets the default font for list boxes, buttons and other controls.
- **Status Bar Font** sets the font for the status bar.

Use builtin image as icon to change your host default settings for the Insight session icon on your desktop.

Use Netscape to View Help Files provides Netscape as your default browser for Insight's **Help** documentation.

Tracing features disabled disables setting tracepoints.

Figure 7: Default style toolbar

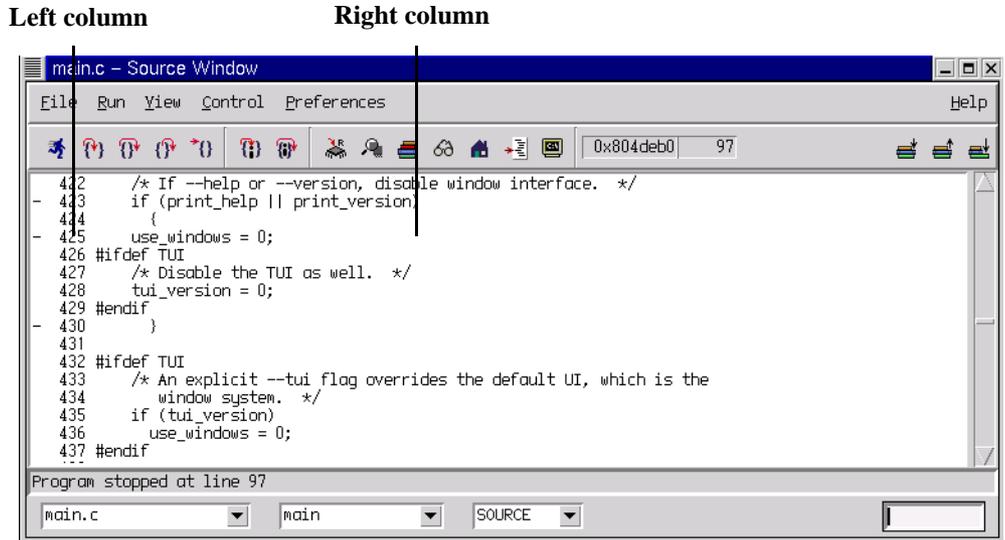
The following descriptions discuss the use of the default debugger toolbar buttons.

-  The **Run** button starts the debugging process for an executable file. If there is no executable open, the Load New Executable dialog displays to open an executable
-  During the debugging process, the **Run** button turns into the **Stop** button to interrupt the debugging. You cannot interrupt some targets; you will instead have to disconnect from the target.
-  The **Step** button steps to next executable line of source code. Also, the **Step** button steps into called functions.
-  The **Next** button steps to the next executable line of source code in the current file. Unlike the **Step** button, the **Next** button steps over called functions.
-  The **Finish** button finishes execution of a current frame. If clicked while in a function, it finishes the function and returns to the line that called the function.
-  The **Continue** button continues execution until a breakpoint, watchpoint or exception is encountered, or until execution completes.
-  The **Registers** button invokes the **Registers** window for viewing or changing register properties for a program's content.
-  The **Memory** button invokes the **Memory** window for displaying and editing the state of memory and addresses.
-  The **Stack** button invokes the **Stack** window for displaying and navigating the current call stack, where each line represents a stack frame.

-  The **Watch Expressions** button invokes the **Watch Expressions** window for entering expressions which will be updated every time that the executable stops.
 -  The **Local Variable** button invokes the **Local Variables** window for displaying all local variables and their structure.
 -  The **Breakpoints** button invokes the **Breakpoints** window for examining breakpoints and changing their settings.
 -  The **Console** button invokes the **Console** window as a command line interface for debugging. `(gdb)` is the prompt.
- | | |
|----------|----|
| 0x401122 | 16 |
|----------|----|
- The left-hand read-only frame displays the program counter (PC) of the current frame.
 - The right-hand read-only frame displays the line number, which contains the program counter.
 -  The **Step assembler** button steps through one assembler machine instruction. Also, the **Step assembler** button steps into subroutines.
 -  The **Next assembler** button steps to the next assembler instruction. The **Next assembler** button then executes subroutines and steps to the next instruction.
 -  The **Down Stack Frame** button moves down the stack frame one level.
 -  The **Up Stack Frame** button moves up the stack frame one level.
 -  The **Go to Bottom of Stack Frame** button moves to the bottom of the stack frame.

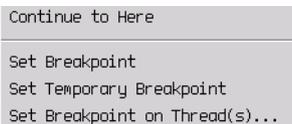
Using the Mouse in the Source Window

The mouse has many uses within the main display pane of the **Source Window**. Divided into two columns (Figure 8), the window's left column extends from the left edge of the display pane to the last character of the line number, while the right column extends from the last character of the line number to the right edge of the display pane. Within each column, the mouse has different effects (see the following descriptions for “Left column functionality for the Source Window” on page 177 and “Right column functionality for the Source Window” on page 178).

Figure 8: Using the mouse in the Source Window

Left column functionality for the Source Window

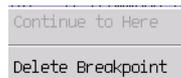
When the cursor is in the left column over an executable line, it appears as a minus sign. When a breakpoint is set at this point, the cursor changes into a circle. A left click sets a breakpoint at the current line; the breakpoint appears as a colored square in place of the minus sign. A left click on any existing or temporary breakpoint removes that breakpoint. A right click on any existing or temporary breakpoint brings up a pop-up menu (Figure 9).

Figure 9: Pop-up menu for setting breakpoints

Continue to Here causes the program to run up to a location, ignoring any breakpoints; like the temporary breakpoint, this menu selection displays as a differently shaded square than a regular breakpoint. When a breakpoint has been disabled, it turns, for instance, from red or orange to black (color settings vary depending on the preferences you set; see also Figure 5 and its accompanying descriptions). **Set Breakpoint** sets a breakpoint on the current executable line; this has the same action as left clicking on the minus sign. **Set Temporary Breakpoint** sets a temporary breakpoint on a current executable line; a temporary breakpoint displays as a differently shaded square than a regular breakpoint, and is automatically removed when hit. **Set Breakpoint on Thread(s)** sets a thread-specific breakpoint at the current location.

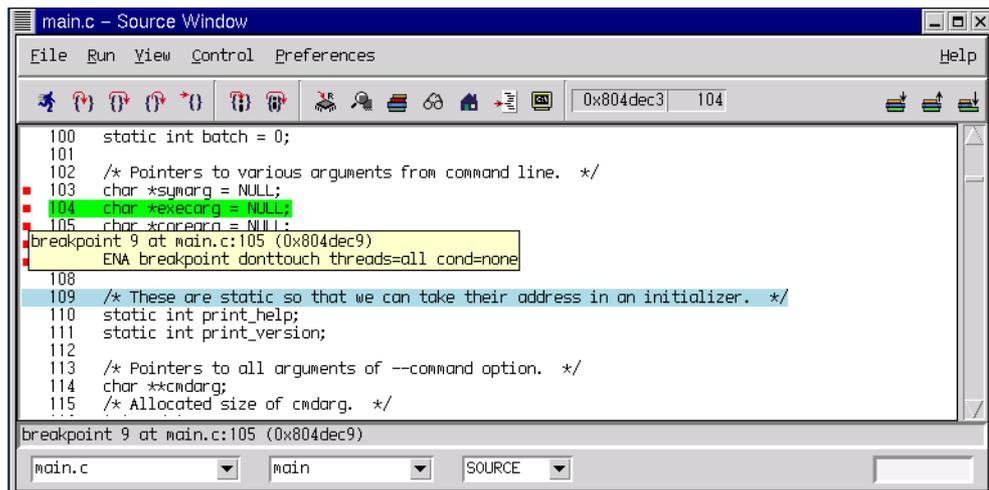
Right-click on a line with a breakpoint to invoke a pop-up menu to delete breakpoints (Figure 10).

Figure 10: Pop-up menu for deleting breakpoints



Delete Breakpoint deletes the breakpoint on the current executable line. This has the same action as left clicking on the colored square; see the description for **Continue to Here** for Figure 9. With the cursor over a line, a breakpoint opens a *breakpoint information balloon*; see Figure 11 for an example of such a tool tip.

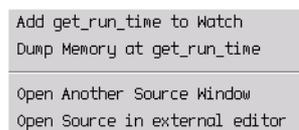
Figure 11: Breakpoint information balloon



Right column functionality for the Source Window

The following documentation discusses the functionality of how the mouse works in the right column of the **Source Window**. With the cursor over a global or local variable, the value of that variable displays. With the cursor over a pointer to a structure or class, view the type of structure or class and the address of the structure or class. Double clicking an expression selects it. Right clicking an expression invokes a pop-up menu (Figure 12).

Figure 12: Pop-up window for expressions



Add *<selected expression>* **to Watch** opens the **Watch Expressions** window (*<selected expression>* in the example was `get_run_time`) and adds a variable expression to the list of expressions in the window. **Dump Memory at**

<selected expression> opens the **Memory** window, which displays a memory dump at an expression. **Open Another Source Window** opens another **Source Window** for displaying a program in an alternate format (see Figure 16 and its accompanying descriptions). **Open Source in external editor** opens the program in an alternate editor, such as the **Source-Navigator Editor** (see “Using the Editor” in *Getting Started*).

Source Window Menus and Display Features

The **Source Window** has the following menu items, many of which correspond to the toolbar buttons (see Figure 7 on page 175).

- **File** has the following menu items. **Edit Source** allows direct editing of the source code. **Open** invokes the **Load New Executable** dialog. **Source** invokes the **Choose GDB Command file** dialog. **Exit** closes the Insight interface.
- **Run** has the following usage. **Attach to Process** attaches thread processes for debugging (see “Using the Processes Window for Threads” on page 197). **Download** downloads an executable to a target. **Run** runs the executable.
- **View** displays the following windows: **Stack** (Figure 20), **Registers** (Figure 21), **Memory** (Figure 22), **Watch Expressions** (Figure 24), **Local Variables** (Figure 29), **Breakpoints** (Figure 33), **Console** (Figure 38), **Function Browser** (Figure 39), and **Processes** (for threads, use the **Threads List** menu item).
- **Control** has the following usage. **Step** steps to next executable line of source code and steps into called functions. **Next** steps to next executable line of source code in the current file and steps over called functions. **Finish** finishes execution of a current frame and, if clicked while in a function, finishes the function and returns to the line that called the function. **Continue** continues execution until a breakpoint, watchpoint or exception is encountered, or until execution completes. **Step Asm Inst** steps through one assembler machine instruction and steps into subroutines. **Next Asm Inst** steps to the next assembler instruction but steps over subroutines.
- **Preferences** has the following usage. **Global** opens **Global Preferences** (Figure 6) for changing how text appears. **Source** opens the **Source Preferences** (Figure 5) to show how colors display.
- **Help** has the following usage. **Help** displays the **Help** window (Figure 42). **About GDB** displays the version number, copyright notice and contact information for Insight to use for GDB.

Below the horizontal scroll bar of the Source Window

There are four display and selection fields below the horizontal scroll bar: the status text box (Figure 13), the file drop-down combo box (Figure 15), the function drop-down combo box (Figure 14) and the code display drop-down list box (Figure 16). At the top of the horizontal scroll bar, text details the current status of the debugger; the status text box in Figure 13 shows “Program stopped at line 19” as

the message.

The **Function Browser** window provides even more powerful tools for locating files and functions within your source code; for more information, see “Using the Function Browser Window” on page 195.

Figure 13: Status text box



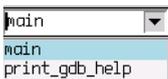
The function drop-down list box (Figure 14) displays all the functions of a selected source (.c) or header (.h) file that an executable uses. Select a function by clicking in the list, or by typing directly into the text field for the function drop-down list box.

Figure 14: Function drop-down combo box



The file drop-down list box (Figure 15) displays the source (.c) and header (.h) files associated with an executable. Select files by clicking the arrow to the right of the drop-down list and then selecting one of the files in the list, or by typing the file’s name directly into the list’s text field.

Figure 15: File drop-down list box



Select how the code in the **Source Window** displays by using the code display drop-down list box (Figure 16).

Figure 16: Code display drop-down list box



The selections in the code display drop-down list box provide the following different ways to display code in the **Source Window**.

- **SOURCE** displays source code.
- **ASSEMBLY** displays assembly code.
- **MIXED** displays both source code *and* assembly code, interspersed within the **Source Window**.
- **SRC+ASM** displays a program’s source and assembly code in separate panes.

Type a character string into the search text box (Figure 17). Press **Enter** to perform a

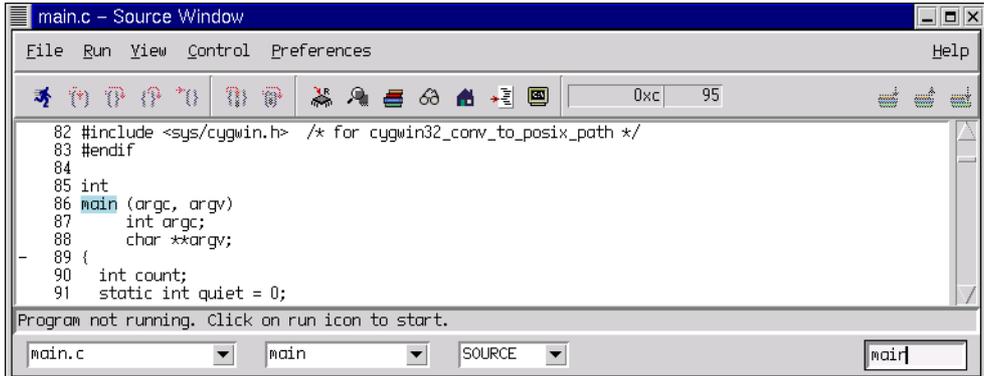
forward search on the source file for the first instance of a specific character string.

Figure 17: Search text box



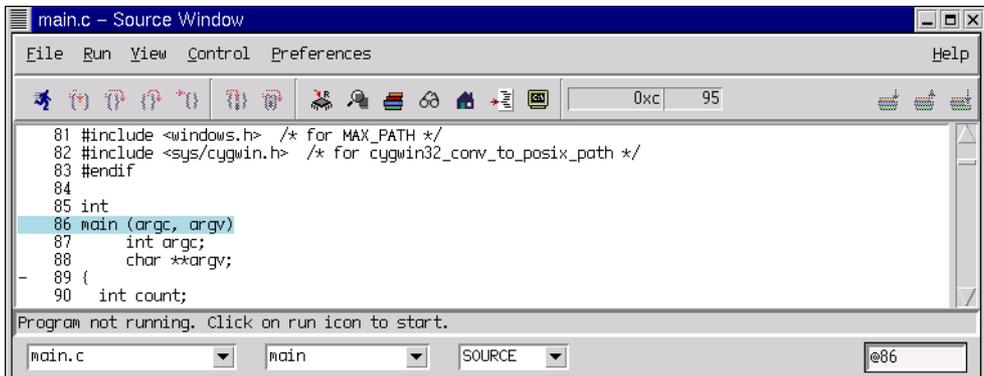
After having specified “main” in the search text box, the example program in Figure 18 shows the jump to a main function.

Figure 18: Searching for a word in source code



Use the **Shift** and **Enter** keys simultaneously to search for the string. Use the **Enter** key or the **Shift** and **Enter** keys to repeat the search. Type “@” with a number in the search text box and press **Enter** to jump to a specific line number in the source code. The example program in Figure 19 shows a jump to the line 86.

Figure 19: Searching for a specific line in source code



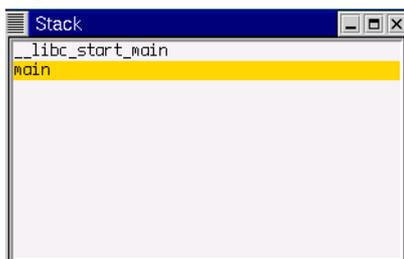
Using the Stack Window

Each time your program performs a function call, information about the call generates. That 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*. When your program stops, you can examine the stack you to see this information.

A stack refers to the layers (TCP, IP, and sometimes others) through which all data passes at both client and server ends of a data exchange. The call stack is the data area or buffer used for storing requests that need to be handled, as in a list of tasks or, specifically, the contiguous parts of the data associated with one call to a specified function in a frame. The frame contains the arguments given to the function, the function's local variables, and the address at which the program is executing.

The **Stack** window displays the current state of the call stack (Figure 20), where each line represents a stack frame; the line with the `main.c` executable had been selected for the example.

Figure 20: Stack window



Click a frame to select or highlight that frame. The **Source Window** automatically shows the source code that corresponds to the selected frame. If the frame points to an assembly instruction, the **Source Window** changes to assembly code; the corresponding source line's background in the **Source Window** also changes to the stack color.

Using the Registers Window

The **Registers** window (Figure 21) dynamically displays registers and their contents.

Figure 21: Registers window

Register	Value	Name	Flags/Status
eax	0x82dda4	st0	0x 3f fe d6 d6 d6 d6 d6 d8 00 {}
ecx	0x804deb0	st1	0x 00 00 00 00 00 00 00 00 00 {}
edx	0x40202234	st2	0x 00 00 00 00 00 00 00 00 00 {}
ebx	0x402051b4	st3	0x 00 00 00 00 00 00 00 00 00 {}
esp	0xbffff95c	st4	0x 00 00 00 00 00 00 00 00 00 {}
ebp	0xbffff978	st5	0x 3f fe 80 00 00 00 00 00 00 {}
esi	0xbffff9a4	st6	0x 3f fe 80 00 00 00 00 00 00 {}
edi	0x1	st7	0x 40 1c f4 24 08 00 00 00 00 {}
eip	0x804deb0	fctrl	0xffff037f
eflags	0x246	fstat	0xffff0000
cs	0x23	ftag	0xffffffff
ss	0x2b	fiseg	0x23
ds	0x2b	fioff	0x80564c6
es	0x2b	foseg	0xffff002b
fs	0x0	fooff	0xbffff6c50
qs	0x0	fop	0x77d

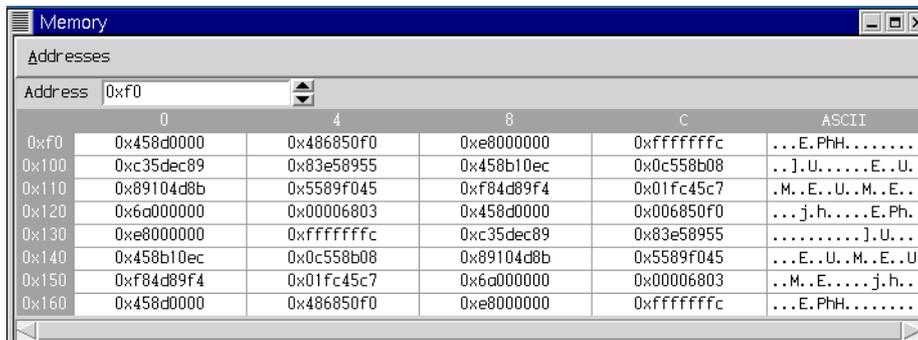
To change the properties of registers, use the following methods.

- To select a register, single left click on it.
- To edit the contents of a register, double click on it. Alternatively, use **Register** → **Edit** to change the contents after selecting a register. Use the **Esc** key to abort the editing.
- Use **Register** → **Format** to invoke another pop-up menu to display the contents of a selected register in **Hex** (Hexadecimal), **Decimal**, **Natural**, **Binary**, **Octal**, or **Raw** formats.
Hex is the default display format. **Natural** format refers to and **Raw** refers to the source format. The other formats are self-explanatory.
- Use **Register** → **Remove from Display** to remove a selected register from the window; all registers will display if you close and reopen the window, unless you have already selected this feature.
- Use **Register** → **Display All Registers** to display all the registers; this menu item is only active when one or more registers have been removed from display.

Using the Memory Window

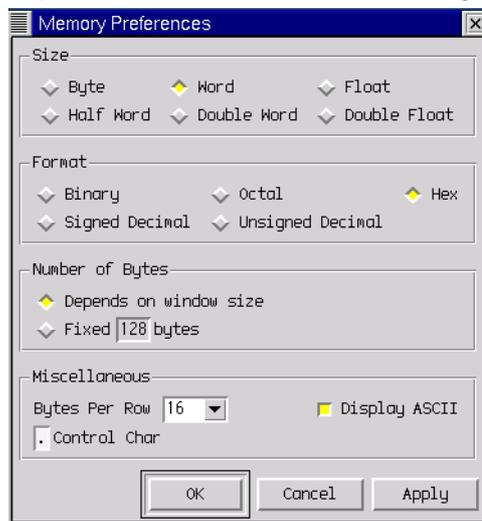
The **Memory** window (Figure 22) dynamically displays the state of memory. Double-click a memory location with the cursor in the window and edit its contents.

Figure 22: Memory window



Use **Addresses** → **Auto Update** to update the contents of the **Memory** window automatically whenever the target's state changes; this is the default setting. Use **Addresses** → **Update Now** to update the **Memory** window's view of the target's memory.

Figure 23: Memory Preferences dialog for the Memory window



Use **Addresses** → **Preferences** to invoke the **Memory Preferences** dialog to set memory options.

- Select the size of the individual cells to display with **Size** options; **Byte**, **Half-Word**, **Word**, **Double-Word**, **Float**, or **Double-Float** are the settings, with

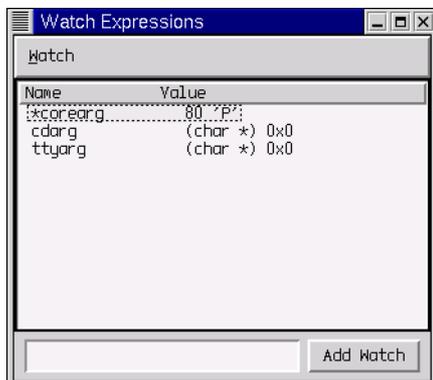
Word being the default selection.

- Select the format of the memory that displays with **Format** options; **Binary**, **Signed Decimal**, **Octal**, **Unsigned Decimal**, or **Hex** (Hexadecimal) are the settings, with **Hex** being the default selection.
- Set the number of bytes to display with **Number of Bytes**, **Depends on Window Size** or **Fixed**. **Depends on Window Size** selection is default.
- Display a string representation of memory with **Miscellaneous**, **Bytes Per Row** or **Display ASCII** selections. **Control Char** displays non-ASCII characters; the default control character is the period (.).

Using the Watch Expressions Window

The **Watch Expressions** window displays the name and current value of user-specified expressions (Figure 24).

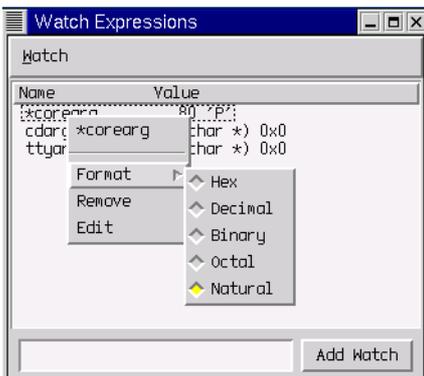
Figure 24: Watch Expressions window



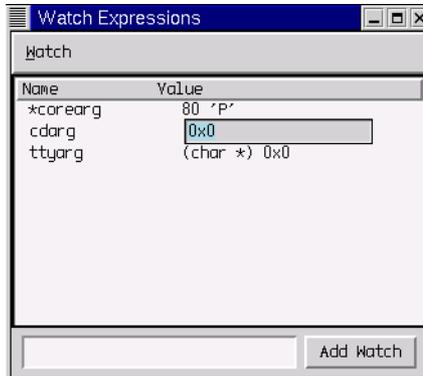
The **Watch Expressions** window has the following functionality.

- Single click on an expression to select it.
- Right click in the display pane, having selected an expression, to invoke an expression-specific **Watch** menu (Figure 25).

Figure 25: Watch menu in the Watch Expressions window



Use **Watch** → **Edit** to edit the value in an expression (an example of an expression capable of being edited is shown in Figure 26). Use the **Esc** key to abort editing.

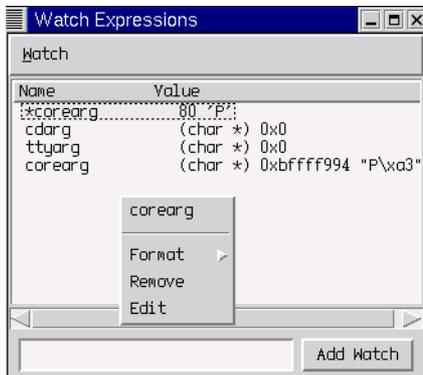
Figure 26: Editing the value in an expression

Use **Watch** → **Format** to invoke another pop-up menu for displaying a selected expression's value in **Hex** (Hexadecimal), **Decimal**, **Binary**, or **Octal** formats; by default, pointers display in hexadecimal with all other expressions as decimal. Use **Watch** → **Remove** to remove a selected expression from the watch list.

Use the text edit field and the **Add Watch** button at the bottom of the window to add registers to the **Watch Expression** window or, by typing register *convenience variables* into the text edit field, add an expression to the watch list (see `corearg` added in Figure 27 with its results in Figure 28).

Figure 27: Using the **Add Watch** button for the **Watch Expressions** window

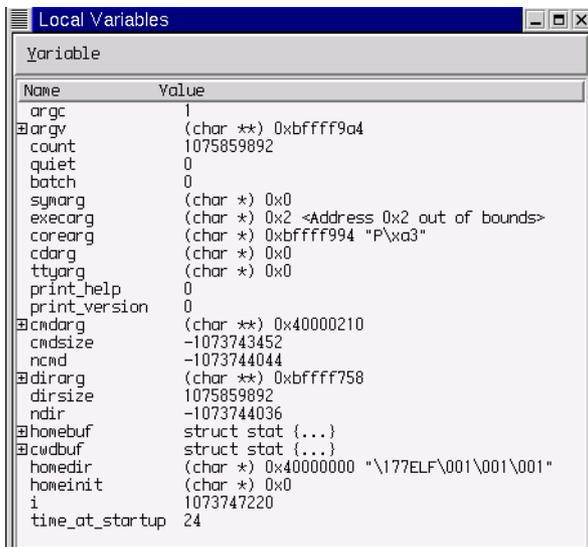
Every register has a corresponding convenience variable. The register convenience variables consist of a dollar sign followed by the register name; `$pc` is the program counter's convenience variable, for example, while `$fp` is the frame pointer's convenience variable. Re-cast other types to which a pointer was cast by typing it in the text edit field. For example, typing `(struct _foo *) bar` in the text edit field, the `bar` pointer is cast as a `struct _foo` pointer. Invalid expressions are ignored.

Figure 28: Results of using **Add Watch** button for the **Watch Expressions** window

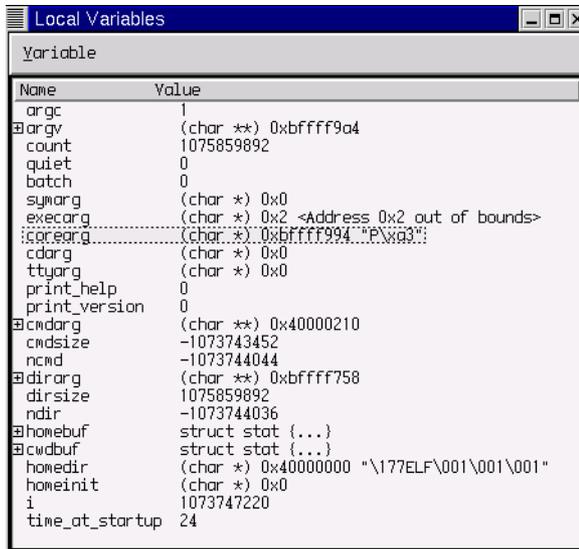
Using the Local Variables Window

The **Local Variables** window displays the current value of all local variables.

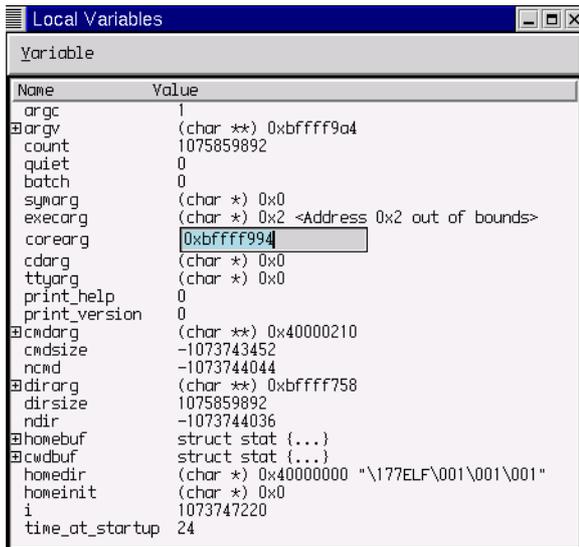
Figure 29: Local Variables window



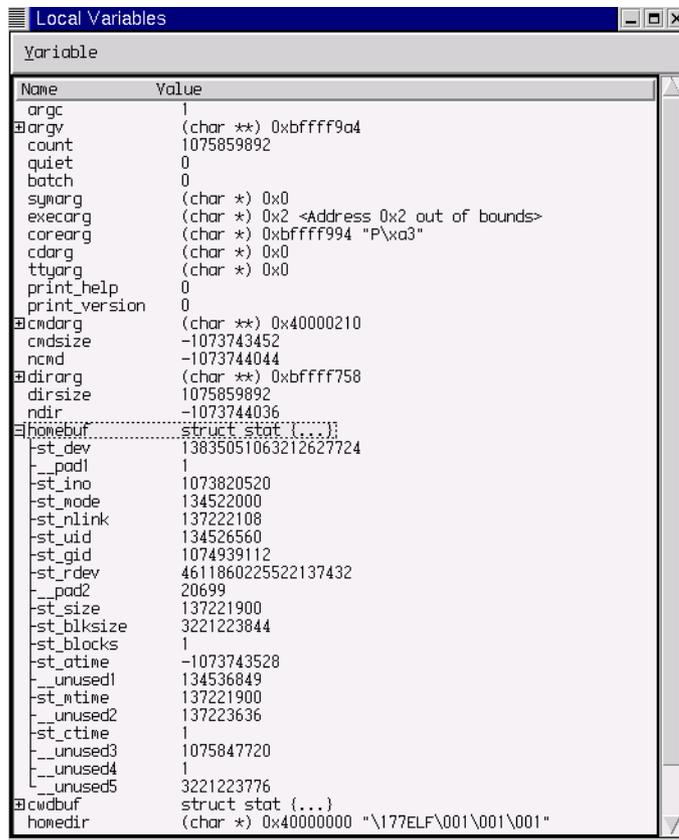
Use **Variable** → **Edit** to change the value of a selected variable that you want edit. Using the Escape key (**Esc**) aborts editing. Use **Variable** → **Format** to invoke another pop-up menu to display a selected variable's value in **Hex** (Hexadecimal), **Decimal**, **Binary** or **Octal** formats. By default, pointers display in hexadecimal and all other expressions as decimal. Single click the mouse with the cursor over a variable in the **Local Variables** window to select the variable (Figure 30).

Figure 30: Selecting a variable

Double click the mouse with the cursor in the **Local Variables** window to edit the variable (Figure 31).

Figure 31: Editing local variables

Single click the mouse with the cursor on the plus sign to the left of a structure variable to see the elements of that structure (compare the variable structure for `homebuf` in Figure 30 with the results in Figure 32). To close the structure elements, click the minus sign to the left of an open structure (compare the variable structure in Figure 32 with what the window had displayed in Figure 30).

Figure 32: Displaying the elements of a variable structure

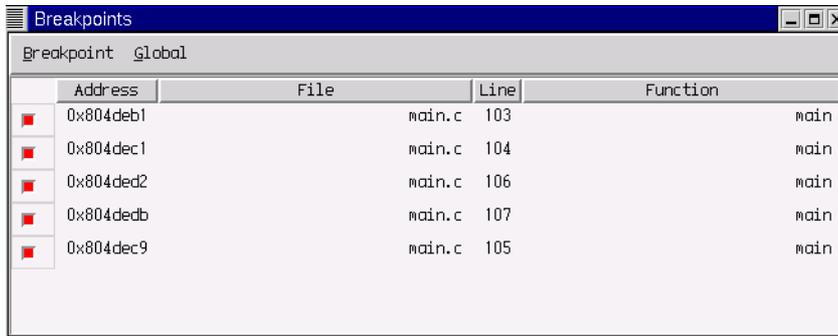
See also “Setting Breakpoints and Viewing Local Variables” on page 202 and Figure 52: “File after changing local variables values” on page 205.

Using the Breakpoints Window

The **Breakpoints** window displays the currently set breakpoints. See Figure 33 for the `main.c` example program breakpoints running in the **Source Window**, and see Figure 36 for the results in the **Source Window**.

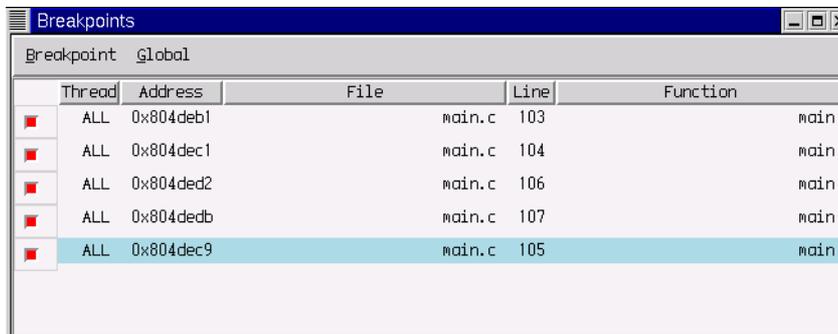
WARNING! Breakpoints and exceptions may not work, especially if debugging C++ code, and the **Breakpoints** window may be inoperative.

Figure 33: Breakpoints window

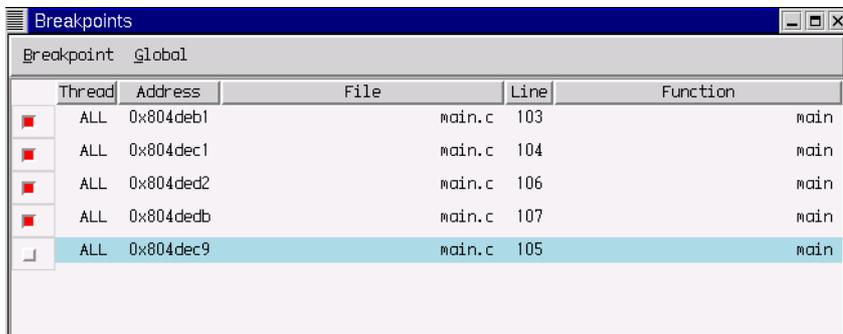


Single click the mouse with the cursor over a check-box for a breakpoint to select that breakpoint (see the breakpoint results for line 105 in Figure 34).

Figure 34: Selecting a breakpoint



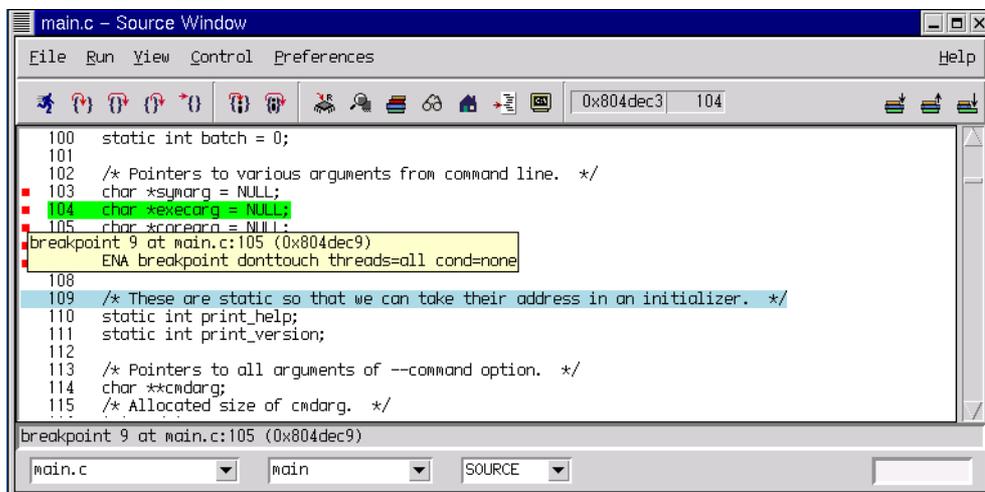
Single click with the mouse with the cursor over a check-box of a breakpoint to disable the breakpoint. The color of the square in the Breakpoint window changes (Line 101 in Figure 35) and the line's breakpoint status changes in the **Source Window**.

Figure 35: Setting temporary breakpoints in the **Breakpoints** window

Using the **Breakpoint** menu for the **Breakpoints** window, toggle the enabled or disabled state of a selected breakpoint. The single check mark between them shows the state of the selected breakpoint. **Remove** removes the selected breakpoint.

Using the **Global** menu for the **Breakpoints** window, **Disable All** disables all breakpoints, **Enable All** enables all breakpoints, and **Remove All** removes all breakpoints.

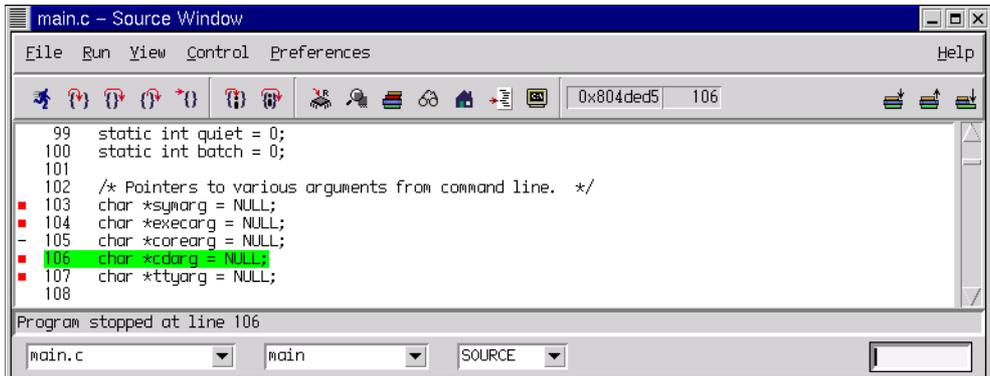
Single click an empty check box of a disabled breakpoint to re-enable a breakpoint (Figure 36). A check reappears and the color of the square in the **Source Window** changes (see line 105 in Figure 37 on page 193).

Figure 36: Results in **Source Window** having enabled a breakpoint

Using the **Breakpoint** menu, toggle between the normal and temporary setting of a selected breakpoint. A normal breakpoint remains valid no matter how many times it is hit. A temporary breakpoint is removed automatically the first time it is hit. A single check mark for either setting shows the state of the selected breakpoint. When a breakpoint is set to temporary, the line in the **Source Window** no longer has a colored square, as shown by comparing Figure 36 with Figure 37, where the breakpoint for

line 105 in the `main.c` example program changed.

Figure 37: Results in Source Window having set a breakpoint as temporary

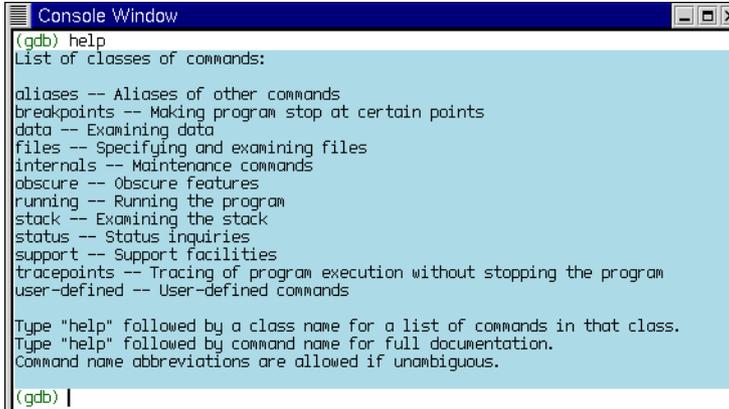


See also Figure 51: “Local Variables window after setting breakpoints” on page 204.

Using the Console Window

To send commands directly to the GDB interpreter, use the **Console** window (Figure 38).

Figure 38: Console window

A screenshot of a terminal window titled "Console Window". The window has a blue title bar and standard window controls (minimize, maximize, close) on the right. The text inside the window is as follows:

```
(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.

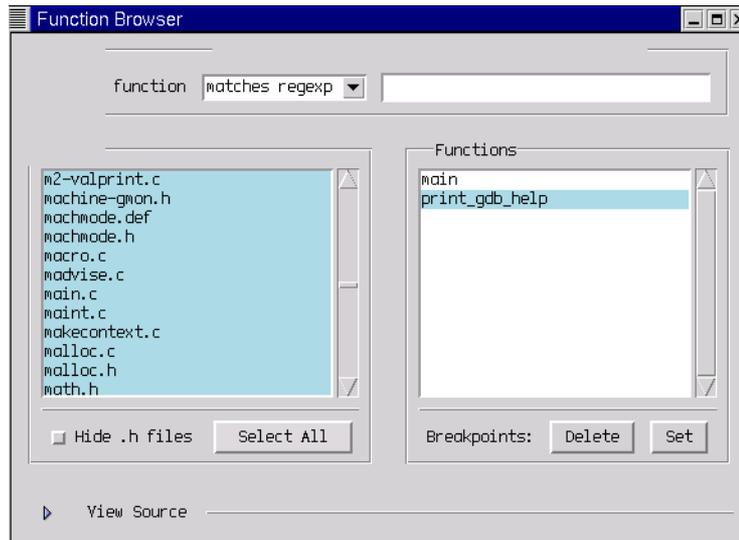
(gdb) |
```

The **Console** window opens with a `(gdb)` prompt for invoking debugging commands. Figure 38 shows the `help` command's available topics when using the **Console** window. For more specific commands, see *Debugging with GDB in GNUPro Debugger Tools*.

Using the Function Browser Window

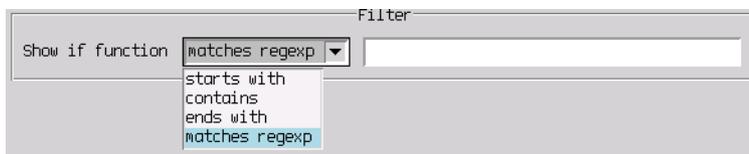
To invoke the **Function Browser** window, select **View** → **Function Browser** from the **Source Window**. The **Function Browser** window has several fields that provide search and browsing capability for source code debugging (Figure 39). Descriptions follow of the **Filter**, **Files**, **Functions** and **View Source** fields.

Figure 39: Function Browser window



The **Filter** group at the top of the **Function Browser** window contains the **Show if function** drop-down list box and a text edit field. **Show if function** allows you to match the character string in the text edit field to its right by any of the four alternatives. Using the **Show if function** drop-down list box (Figure 40), **starts with** shows functions that start with the character string in the text edit field entry, **contains** shows functions that contain the character string in the text edit field entry, **ends with** shows functions that end with the character string in the text edit field entry, **matches regexp** makes the search routines use regular expression matching (for example, searching for “`^[ab].*`” matches all functions starting with either a or b letters).

Figure 40: Show if function drop-down list box



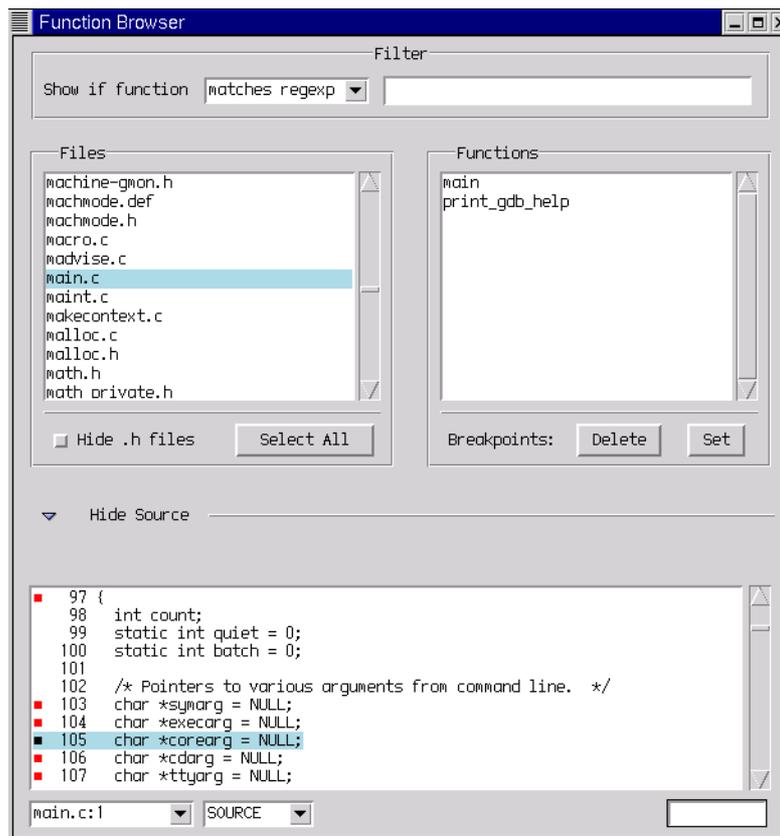
The **Files** group limits the search to highlighted files. Click individual file names to select or deselect that file. The list of matching files refreshes when any search parameter changes. **Hide .h files**, if checked, disallows .h header files to display.

Select All selects all listed files.

The **Functions** group matches all functions in the selected file(s). **Breakpoints** has two available buttons, **Delete** or **Set**; **Delete** removes a breakpoint previously set at the first executable line of the selected function, while **Set** sets a breakpoint at the first executable line of the selected function. Both of these will work on any and all selected functions in the listing. If all functions are selected, they all get or lose a breakpoint.

View Source/Hide Source allows you to toggle between displaying or hiding a file in a source browser (Figure 41); the source browser has the same functionality as when using the **Source Window**.

Figure 41: Function Browser window with source browser



There are four display and selection fields below the horizontal scroll bar (the same functionality as using the **Source Window**): the status text box (Figure 13), the function drop-down combo box (Figure 14) and the code display drop-down list box (Figure 16); see the figures and their accompanying explanations for specific information.

Using the Processes Window for Threads

The **Processes** window dynamically displays the state of currently running threads.

WARNING! Threads support is not available for all targets.

The **Processes** window will display a list of threads and/or processes of an executable that you are debugging. The exact contents are specific to each operating system. The first column is the thread number, used internally by the debugger to track the thread. This number is also used by the command line interface (in the **Console** window) when referring to threads. The rest of the columns are dependent on information coming from the operating system.

The **Source Window** displays the current location and source for a *current* thread (or process). To change the current thread, click on the desired thread in the **Processes** window and the debugger will switch contexts, updating all windows. The current thread will highlight.

Having set a breakpoint on a line or function, stop execution and return control to the debugger for every thread that hits a set location. To set a breakpoint on a specific thread or threads, use the **Source Window**. See also “Setting Breakpoints and Viewing Local Variables” on page 202 and “Setting Breakpoints on Multiple Threads” on page 206.

Using the Help Window

Invoke the **Help** window (Figure 42) using the **Help** menu from the **Source Window** to get HTML-based navigable help by topic.

Figure 42: Help window showing the help topic's index



The **Help** window has two menus: **File** and **Topics**.

The **File** menu makes the following options functional: **Back** moves back one HTML help page, relative to previous forward page movements; **Forward** moves forward one HTML help page, relative to previous back page movement; **Home** returns to the main HTML help “Table of Contents” page; **Close** closes the **Help** window.

The **Topics** menu displays information for each menu item. Content changes in the **Help** window to represent a selected topic. The first menu item, **index**, returns to the main **Help** window (Figure 42). The second item, **Attach Dialog**, is only for a host system’s use, when attaching to another debugging process, and *not* for use by embedded targets. The remaining menus document the Insight windows: **Stack** (Figure 20), **Registers** (Figure 21), **Memory** (Figure 22), **Watch Expressions** (Figure 24), **Local Variables** (Figure 29), **Breakpoints** (Figure 33), **Console** (Figure 38), **Function Browser** (Figure 39), and **Threads** (for the **Processes** window when working with threads; the window contents are dependent on the operating system in use).

2

Examples of Debugging with Insight

The following documentation contains examples of debugging session procedures for using Insight; the content assumes familiarity with GDB and its main debugging procedures.

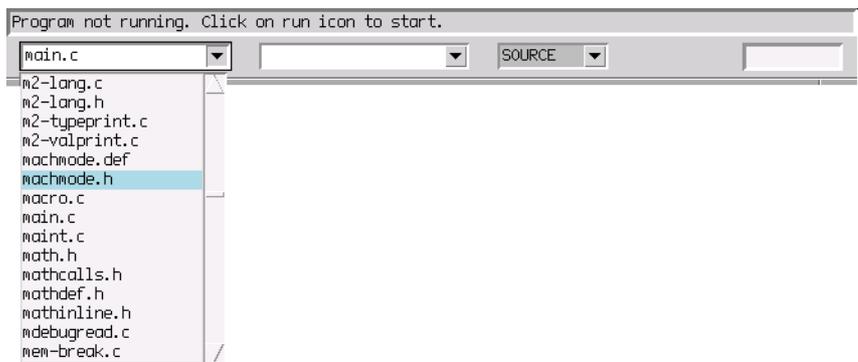
- “Selecting and Examining a Source File” on page 200
- “Setting Breakpoints and Viewing Local Variables” on page 202
- “Setting Breakpoints on Multiple Threads” on page 206

Selecting and Examining a Source File

To select a source file, or to specify what to display when examining a source file when debugging, use the following process.

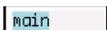
1. Select a source file from the file drop-down list, at the bottom left of the **Source Window** (`main.c` in the example in Figure 43).

Figure 43: Source file selection



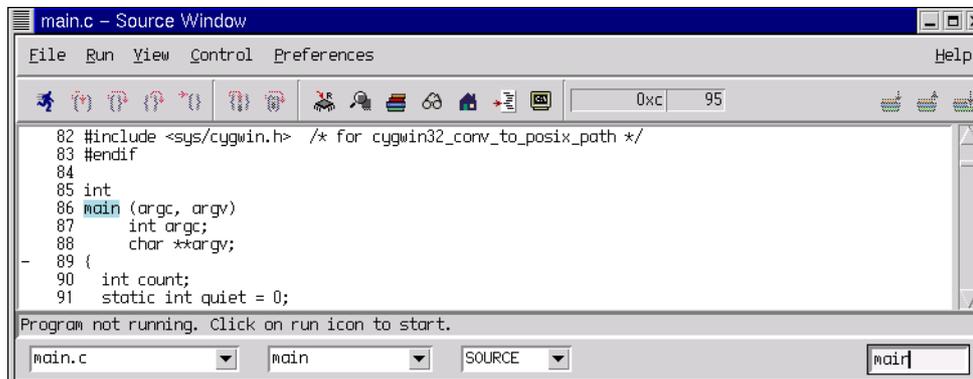
2. Select a function from the function drop-down list to the right of the file drop-down list, or type its name in the text field above the list to locate a function (in Figure 45, see the executable line 86, where the `main` function displays).
3. Type a character string into the search text box (Figure 44).

Figure 44: Search text box



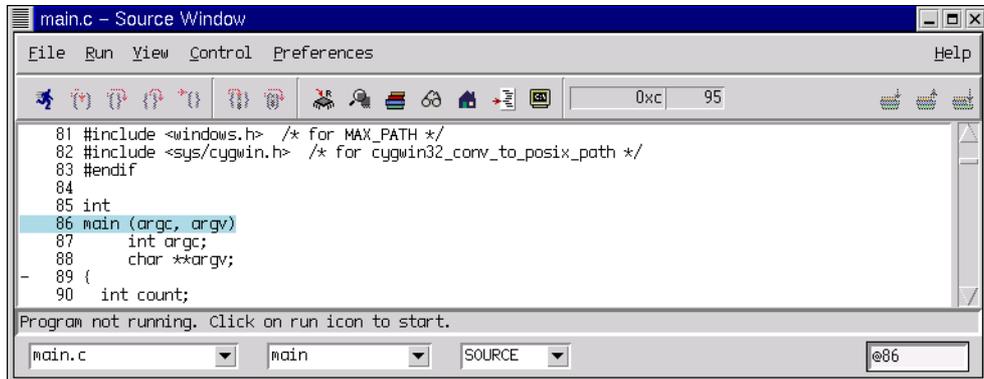
4. Press **Enter** to perform a forward search on the source file for the first instance of a specific character string. After having specified `main` in the search text box, the example program in Figure 45 shows the jump to a `main` function.

Figure 45: Searching for a word in source code



5. Use the **Shift** and **Enter** keys simultaneously to search for the string. Use the **Enter** key or the **Shift** and **Enter** keys to repeat the search. Type “@” with a number in the search text box and press **Enter** to jump to a specific line number in the source code. The example program in Figure 46 shows a jump to the line 86.

Figure 46: Searching for a specific line in source code



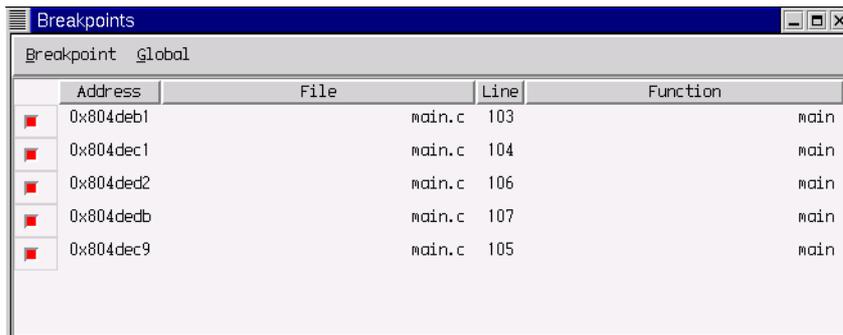
Setting Breakpoints and Viewing Local Variables

A breakpoint can be set at any executable line in a source file. Executable lines are marked by a minus sign in the left margin of the **Source Window**. When the cursor is in the left column and it is over an executable line, it changes into a circle. When the cursor is in this state, a breakpoint can be set.

The following exercise steps you through setting four breakpoints in a function, as well as running the program and viewing the changing values in the local variables.

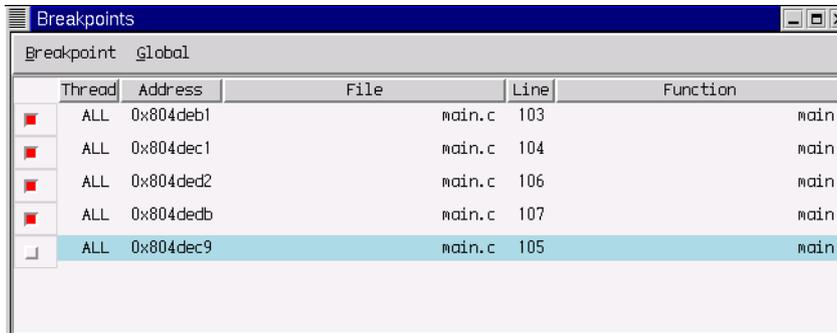
1. With the **Source Window** active and the `main.c` source file open, the cursor was placed over the minus sign on line 6.
2. When the minus sign changes into a circle, click the left mouse button; this sets the breakpoint, indicated by a colored square.
3. Click on a breakpoint to remove the breakpoint.
4. Repeat the process to set breakpoints at specific lines.
5. Open the **Breakpoints** window (Figure 47).

Figure 47: Breakpoints window



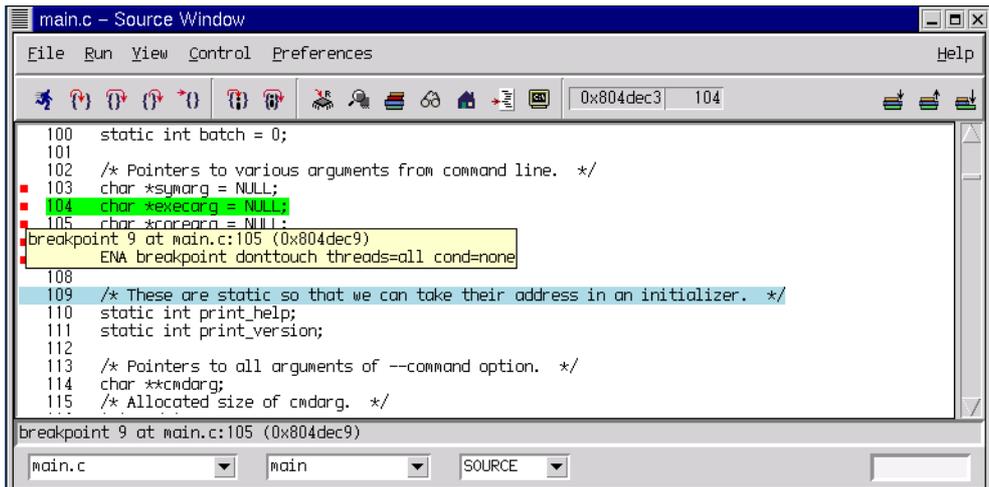
Address	File	Line	Function
0x804deb1	main.c	103	main
0x804dec1	main.c	104	main
0x804ded2	main.c	106	main
0x804dedb	main.c	107	main
0x804dec9	main.c	105	main

6. Click the check box for a line to set a breakpoint in an executable. The box's color changes and the square's color of the line in the **Source Window** changes (Figure 48). This color change indicates a disabling of the breakpoint.

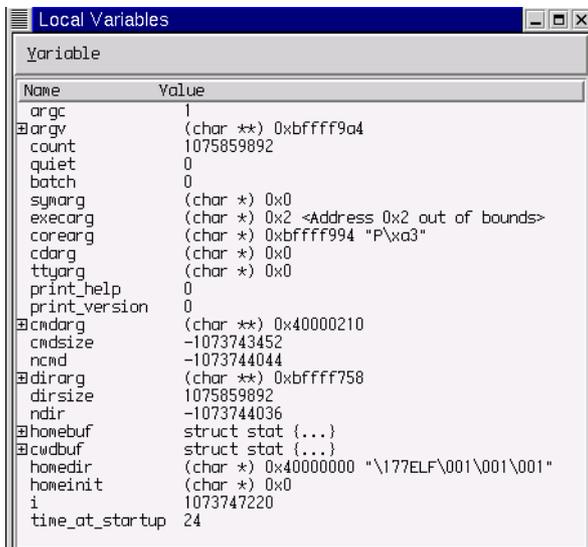
Figure 48: Disabling a breakpoint in **Breakpoints** window

Re-enable the breakpoint at the line by clicking the check box in the **Breakpoints** window.

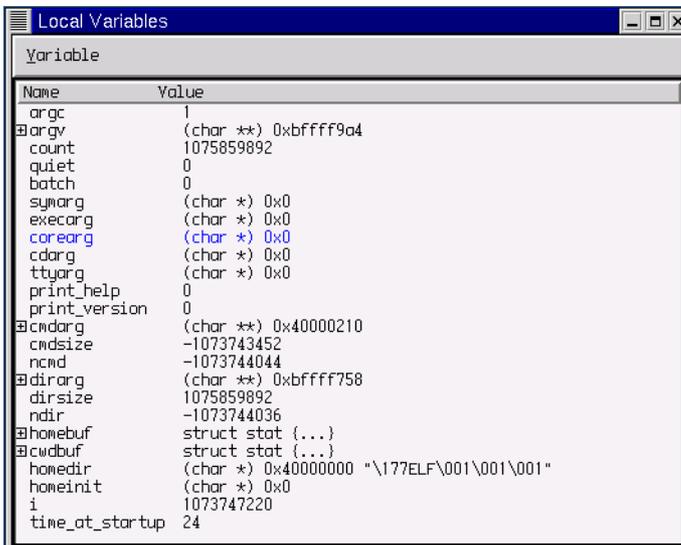
- Click the **Run** button on the tool bar to start the executable (see “Run button” on page 172). The program runs until it hits the first breakpoint. The color bar on the line changes color, indicating that the program is running (see settings in Figure 47 changed in Figure 48, and the **Source Window** in Figure 49: “Results of setting breakpoints at line 105” on page 203, after debugging stopped.).

Figure 49: Results of setting breakpoints at line 105

- Open the **Local Variables** window (Figure 50), by clicking the **Local Variables** button in the tool bar. The window displays the initial values of the variables.

Figure 50: Local Variables window

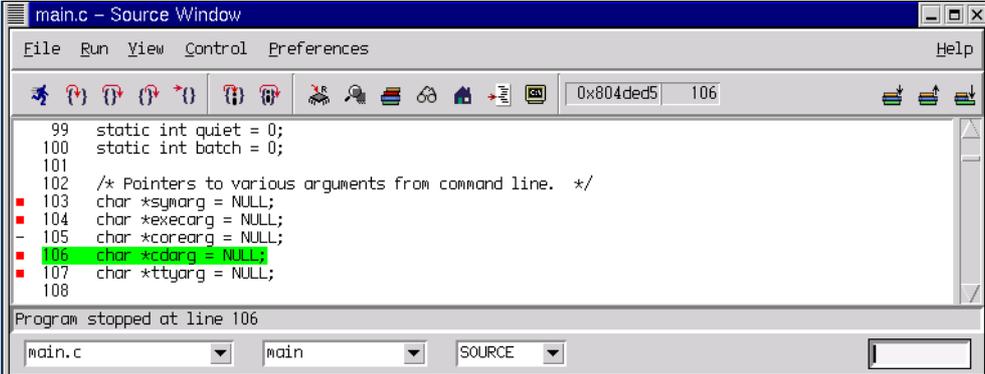
- Click the **Continue** button in the tool bar to move to the next breakpoint. The variables that have changed value turn color in the **Local Variables** window (see results in Figure 51 for line 105 in the `main.c` example).

Figure 51: Local Variables window after setting breakpoints

- Click the **Continue** button two more times to step through the next two breakpoints and notice that the values of the local variables change (compare results from the `main.c` example program in Figure 49 and results in Figure 52). Repeat with the **Continue** button to step through breakpoints and notice their

values change.

Figure 52: File after changing local variables values



Setting Breakpoints on Multiple Threads

With Insight processing in a multi-thread environment, select threads and set breakpoints on one or more threads when debugging.

WARNING! Multiple thread functionality does not work similarly on all embedded targets. When debugging C++ code, for instance, breakpoints and exceptions may not work on multiple threads.

A process can have multiple threads running concurrently, each performing a different task, such as waiting for events or something time-consuming that a program doesn't need to complete before resuming. When a thread finishes its job, the debugger suspends or destroys the thread running in the debugging process.

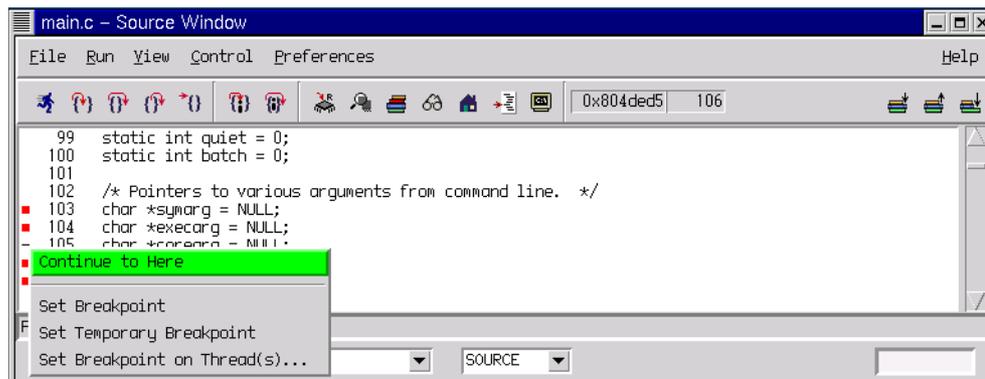
The thread debugging facility allows you to observe all threads while your program runs. However, whenever the debugging process is active, one thread in particular is always the focus of debugging. This thread is called the *current thread*.

The precise semantics of threads and the use of threads differs depending on operating systems.

In general, the threads of a single program are like multiple processes—except that they share one address space (that is, they can all examine and modify the same variables). Additionally, each thread has its own registers and execution stack, and perhaps private memory.

1. In the **Source Window**, right click on an executable line without a breakpoint to open the breakpoint pop-up menu (see Figure 53).

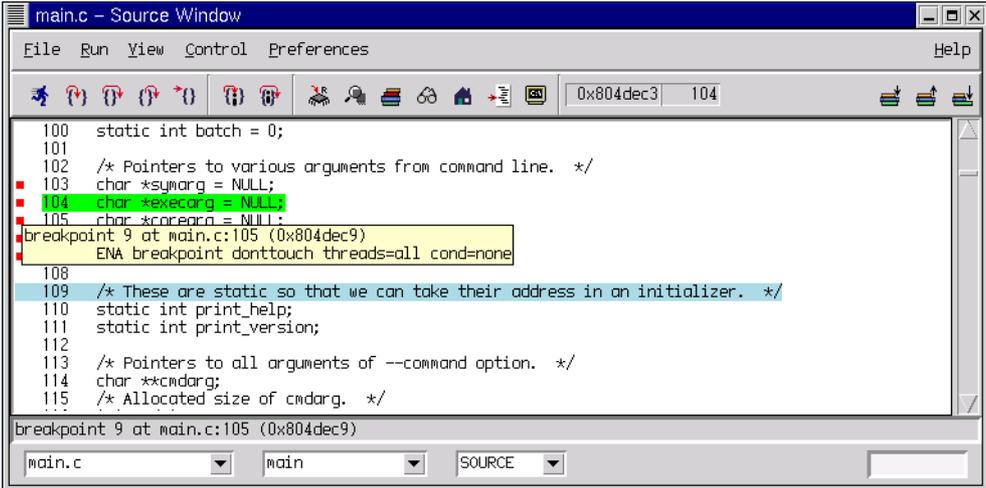
Figure 53: Breakpoint pop-up menu in the **Source Window**



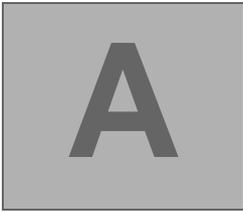
2. Select the **Set Breakpoint on Thread(s)** menu item. The **Processes** window displays.
3. By clicking on specific breakpoints, select one or more threads. A breakpoint sets in the **Source Window** at the executable line only for the selected threads. Having selected threads, the results display in the **Processes** window. With the cursor over a breakpoint at line 105 in the sample program in the **Source Window**, a

breakpoint information balloon displayed to show where the selected thread begins (Figure 54).

Figure 54: Breakpoint balloon with thread information in **Source Window**



Appendixes



Using GDB under GNU Emacs

GNU Emacs allows you to use, to view and to edit the source files for the program you are debugging with GDB. The following documentation provides information especially for use with the Emacs text editor.

- “Emacs Considerations with GDB” (below)
- “Keystroke Sequences for GDB with Emacs” on page 212

Emacs Considerations with GDB

Using GDB under Emacs is just like using GDB normally except for the following considerations.

- 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.

Some of the following material uses the convention laid out in the *GNU Emacs Manual*[†]. **Meta-** signifies using the Meta key (the diamond key, which is only on some UNIX keyboards) or the **Alt** key on other keyboards, followed by the

[†] Documentation available from the Free Software Foundation (ISBN 1-882214-03-5).

specified letter. **Ctrl-** signifies using the **Ctrl** key in sequence with a specified letter. Any other input will be signified by code (as in something typed onscreen like the `gdb` command).

To use the Emacs interface, use the command **Meta-x** and type `gdb` then give the executable file you want to debug as an argument; GDB starts as a subprocess of Emacs, with input and output through a newly created Emacs buffer.

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, **Ctrl-c**, **Ctrl-c** for an interrupt, or with **Ctrl-c**, **Ctrl-z** to stop a debugging process.

- 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. However, Emacs does not get enough information from GDB to locate the source files in this situation; to avoid this problem, either start GDB from the directory where your program resides, or specify an absolute file name when using the **Meta-x** `gdb` argument.

A similar confusion can result if you use the GDB file command to switch to debugging a program in some other directory, with an existing GDB buffer in Emacs.

By default, using the keystroke sequence, **Meta-x**, with the input, `gdb`, calls the GDB program. If you need to call GDB by a different name (for example, if you keep several configurations with different names) you can set the Emacs variable, `gdb-command-name`. For example, make Emacs instead call the `mygdb` program, using the input, `setq gdb-command-name "mygdb"` (preceded by using the **Esc** key twice, or by typing in the `*scratch*` buffer, or in your `.emacs` file).

Keystroke Sequences for GDB with Emacs

In the GDB I/O buffer, you can use the following keystroke sequences of Emacs

commands in addition to the standard Shell mode commands.

Ctrl-h, m

Describe the features of Emacs' GDB mode.

Meta-s

Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

Meta-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.

Meta-i

Execute one instruction, like the GDB `stepi` command; update display window accordingly.

Meta-x, gdb-nexti

Execute to next instruction, using the GDB `nexti` command; update display window accordingly.

Ctrl-c, Ctrl-f

Execute until exit from the selected stack frame, like the GDB `finish` command.

Meta-c

Continue execution of your program, like the GDB `continue` command.

In Emacs version 19, this command uses the keystroke sequence, **Ctrl-c, Ctrl-p**.

Meta-u

Go up the number of frames indicated by the numeric argument (see “Numeric Arguments” in *GNU Emacs Manual*), like the GDB `up` command.

In Emacs version 19, use the keystroke sequence, **Ctrl-c, Ctrl-u**.

Meta-d

Go down the number of frames indicated by the numeric argument, like the GDB `down` command.

In Emacs version 19, use the keystroke sequence, **Ctrl-c, Ctrl-d**.

Ctrl-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 using the keystroke sequence, **Ctrl-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 using the keystroke sequence, **Ctrl-x, &** before they are inserted. A numeric argument to **Ctrl-x, &** indicates that you wish special

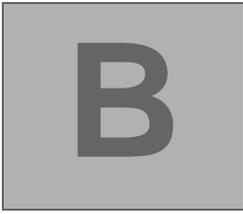
formatting, and also acts as an index to pick an element of the list. If the list element is a string, the number to be inserted is formatted using the Emacs function `format`; otherwise the number is passed as an argument to the corresponding list element.

In any source file, using the keystroke sequence, **Ctrl-x, SPACEBAR**, and typing `(gdb-break)`, tells GDB to set a breakpoint on the source line point.

If you accidentally delete the source-display buffer, an easy way to get it back is to use 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; 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.



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. In any case, the principal function of a bug report is to help the entire GNU community by making the next version of GDB work better. Bug reports are your contribution to the maintenance of GDB. See the following documentation for information for reporting GDB bugs.

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.
- 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.

How to Report Bugs

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug. A number of companies and individuals offer support for GNU products. If you obtained GDB from a support organization, we recommend you contact that organization first. You can find contact information for many support companies and individuals in the file `etc/SERVICE` in the GNU Emacs distribution. In any event, we also recommend that you send bug reports for GDB to one of these addresses:

```
bug-gdb@prep.ai.mit.edu  
{ucbvax|mit-eddie|uunet}!prep.ai.mit.edu!bug-gdb
```

Do not send bug reports to `info-gdb` or to `help-gdb` or to any newsgroups. Most users of GDB do not want to receive bug reports. Those who do have arranged to receive `bug-gdb`.

The `bug-gdb` mailing list has a `gnu.gdb.bug` newsgroup which serves as a repeater. The mailing list and the newsgroup carry exactly the same messages. Often people think of posting bug reports to the newsgroup instead of mailing them. This appears to work, but it has one problem which can be crucial: a newsgroup posting often lacks a mail path back to the sender. Thus, if we need to ask for more information, we may be unable to reach you. For this reason, it is better to send bug reports to the mailing list. As a last resort, send bug reports on paper to:

```
GNU Debugger Bugs  
Free Software Foundation Inc.  
59 Temple Place Suite 330  
Boston, MA 02111-1307 USA
```

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 if it is new to us. Therefore, always write your bug reports on the assumption that the bug has not been reported previously.

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

hide the sender to *report bugs properly*.

To enable us to fix the bug, you should include all the following 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 GDB.
- What compiler (and its version) was used to compile the program you are debugging.
- 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. Your copy might crash and others 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.

- If you wish to suggest changes to the GDB source, send us context diffs. If you even discuss something in the GDB source, refer to it by context, not by line number.

The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

The following 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.

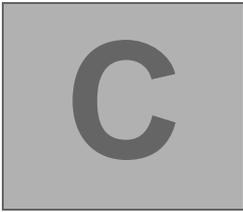
If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

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.



Command Line Editing

The following text describes command line editing interface using Readline library.

- “Readline Interaction” on page 220
- “Readline init File” on page 223
- “Bindable Readline Commands” on page 230
- “Readline in vi Mode” on page 236

The text, **C-k**, is read as “Control K” and describes the command to produce when using the **Ctrl** and the **k** keys together. The text, **M-k**, is read as “Meta K” and describes the command to produce when using the **Meta** key (the key with a diamond), and the **k** key. If you do not have a **Meta** key, the identical keystroke can be generated using the **Alt** key, and then, **k**. Either process is known as “meta-fying the **k** key.” **M-C-k** is read as Meta Control K.

IMPORTANT! The hyphen characters and the comma characters are not a part of the keystroke sequence to type in the following documentation’s descriptions of Readline usage.

In addition, several keys have their own names. Specifically, **Delete**, **Esc**, **LFD** (linefeed), **SPACEBAR**, **Return**, and **Tab** all stand for themselves when seen in this text, or in an `init` file (see “Readline init File” on page 223 for more information).

Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply use **Return**. You do not have to be at the end of the line to use **Return**; the entire line is accepted regardless of the location of the cursor within the line.

See the following documentation for more details.

- “Readline Bare Essentials” on page 220
- “Readline Movement Commands” on page 221
- “Readline Killing Commands” on page 221
- “Searching for Commands in the History” on page 222

Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use the erase tools to delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can use **C-b** to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with **C-f**.

When you add text in the middle of a line, you will notice that characters to the right of the cursor get pushed over to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor get pulled back to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

C-b

Move back one character.

C-f

Move forward one character.

Delete

Delete the character to the left of the cursor.

C-d

Delete the character underneath the cursor.

Printing characters

Insert itself into the line at the cursor.

C-_

Undo the last thing that you did. You can undo all the way back to an empty line.

Readline Movement Commands

The previous commands are the most basic possible keystrokes that you need in order to do editing of the input line. Other commands have been added in addition to **C-B**, **C-F**, **C-D**, and **Delete**, as in the following movement commands.

C-a

Move to the start of the line.

C-e

Move to the end of the line.

M-f

Move forward a word.

M-b

Move backward a word.

C-l

Clear the screen, reprinting the current line at the top.

Notice how **C-f** moves forward a character, while **M-f** moves forward a word. A loose convention is that control keystrokes operate on characters while meta keystrokes operate on words.

Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* it back into the line. If the description for a command says that it kills text, then you can be sure that you can get the text back in a different (or the same) place later. The following is the list of commands for killing text.

C-k

Kill the text from the current cursor position to the end of the line.

M-d

Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

M-Delete

Using the **Meta** key and the **Delete** key, kill from the cursor to the start of the previous word, or if between words, to the start of the previous word.

C-w

Kill from the cursor to the previous whitespace.

This is different than **M-Delete** because the word boundaries differ.

And, here is how to yank the text back into the line.

C-y

Yank the most recently killed text back into the buffer at the cursor.

M-y

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is **C-y** or **M-y**.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it in one clean sweep. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might use **M-- C-k**.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first digit you type is a minus sign (-), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the **C-d** command an argument of 10, you could use the keystroke sequence, **M-1, 0, C-d**.

Searching for Commands in the History

Readline provides commands for searching through the command history for lines containing a specified string.

There are two search modes: incremental and non-incremental.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, Readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. The **Esc** key is used to

terminate an incremental search. **C-j** will also terminate the search. **C-g** will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line. To find other matching entries in the history list, type **C-s** or **C-r** as appropriate. This will search backward or forward in the history for the next entry matching the search string typed up to that point. Any other key sequence bound to a Readline command will terminate the search and execute that command. For instance, using the **Return** key will terminate the search and accept the line, thereby executing the command from the history list.

Non-incremental searches read the entire search string before starting to search for matching history lines. The search string may be typed by the user or be part of the contents of the current line.

Readline `init` File

Although the Readline library comes with a set of Emacs-like key bindings, installed by default, it is possible that you would like to use a different set of key bindings. You can customize programs that use Readline by putting commands in an `inputrc` file in a home directory. `~/.inputrc` is the name of this file.

The following documentation describes more about the `init` file for Readline.

- “Readline `init` Syntax” on page 223
- “Variable Settings for Readline” on page 224
- “Key Bindings for Readline” on page 225

When a program which uses the Readline library starts up, the `~/.inputrc` file is read, and the key bindings are set.

In addition, the **C-x**, **C-r** command re-reads this `init` file, thus incorporating any changes that you might have made to it.

Readline `init` Syntax

The following documentation describes the `init` syntax for Readline’s `~/.inputrc` file.

- “Variable Settings for Readline” on page 224
- “Key Bindings for Readline” on page 225

Variable Settings for Readline

You can modify the run-time behavior of Readline by altering the values of variables in Readline using the following `set` command within the `init` file.

```
set editing-mode vi
```

The following discussion explains how to change from the default Emacs-like key binding to use `vi` line editing commands. A great deal of run-time behavior is changeable with the following variables.

`bell-style`

Controls what happens when Readline wants to signal a change (“ringing the terminal bell”). If set to `none`, Readline never provides a signal. If set to `visible`, Readline uses a visible signal, like a blinking cursor, if one is available. If set to `audible` (the default), Readline uses only the audible signal.

`comment-begin`

The string to insert at the beginning of the line when the `insert-comment` command is executed. `#` is the default value.

`completion-ignore-case`

If set to `on`, Readline performs filename matching and completion in a case-insensitive fashion. `off` is the default value.

`completion-query-items`

The number of possible completions that determines when the user has preferences for possible completion of commands. If the number of possible completions is greater than this value, Readline will ask the user whether or not to make them viewable; otherwise, they are simply listed. The default limit is 100.

`convert-meta`

If set to `on`, Readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prepending an **Esc** character, converting them to a meta-prefixed key sequence. `on` is the default value.

`disable-completion`

If set to `on`, Readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to `self-insert`. `off` is the default.

`editing-mode`

The `editing-mode` variable controls which default set of key bindings is used. By default, Readline starts up in Emacs editing mode, where the keystrokes are as for Emacs. This variable can be set to either `emacs` or `vi`.

`enable-keypad`

When set to `on`, Readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys. `off` is the default.

`expand-tilde`

When set to `on`, tilde expansion is performed when Readline attempts word completion. `off` is the default.

`horizontal-scroll-mode`

This variable can be set to either `on` or `off`. Setting it to `on` means that the text of the lines being edited will scroll horizontally on a single screen line when they are longer than the width of the screen, instead of wrapping onto a new screen line. `off` is the default.

`keymap`

Sets Readline's idea of the current keymap for key binding commands. Acceptable keymap names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default, `keymap`.

`mark-directories`

When set to `on`, completed directory names have a slash appended. `on` is the default.

`mark-modified-lines`

This variable when set to `on`, says to display an asterisk (*) at the starts of history lines which have been modified. This variable is `off` by default.

`input-meta`

If set to `on`, Readline will enable eight-bit input (it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The default value is `off`. The name, `meta-flag`, is a synonym for this variable.

`output-meta`

If set to `on`, Readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. `off` is default.

`print-completions-horizontally`

If set to `on`, Readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen. `off` is default.

`show-all-if-ambiguous`

This alters the default behavior of the completion functions. If set to `on`, words having more than one possible completion cause the matches to be listed immediately instead of ringing the bell. The default value is `off`.

`visible-stats`

If set to `on`, a character denoting a file's type is appended to the filename when listing possible completions. `off` is default.

Key Bindings for Readline

The syntax for controlling key bindings in the `~/.inputrc` file is simple. First you have to know the name of the command that you want to change. Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the `~/.inputrc` file. The name of the key can be expressed in different ways, depending on which is most comfortable for you. `keyname` and `keyseq` are examples.

keyname: *function-name* OR *macro*

keyname signifies the name of a key in English. The following text serves as example.

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

For instance, **C-U** is bound to the function, `universal-argument`, and **C-O** is bound to run the macro expressed on the right hand side (that is, to insert the text `>&output` into the line).

"keyseq": *function-name* OR *macro*

keyseq differs from *keyname* in that strings denoting an entire key sequence can be specified. The key sequence must be indicated in double quotes. GNU Emacs-style key escape sequences can be used, such as in the following examples.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[1l~": "Function Key 1"
```

For instance, **C-U** is bound to the function, `universal-argument`; **C-X**, **C-R** is bound to the function, `reread-init-file`, and **Esc-[1, 1, ~** is bound to insert the text, `Function Key 1`.

The following GNU Emacs style escape sequences are available when specifying key sequences.

```
\C- Control prefix
\M- Meta prefix
\e An escape character
\\ Backslash
\" Double quote
\' Single quote
```

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available.

```
\a Alert (bell)
\b Backspace
\d Delete
```

<code>\f</code>	form feed
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	horizontal tab
<code>\v</code>	vertical tab
<code>\nnn</code>	the character whose ASCII code is the octal value, <code>nnn</code> (one to three digits)
<code>\xnnn</code>	the character whose ASCII code is the hexadecimal value <code>nnn</code> (one to three digits)

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes in the previous descriptions are expanded. Backslash will quote any other character in the macro text, including " and ' (single-quote). For example, the following binding will make `C-x \` insert a single backslash into the line:

```
"\C-x\" : "\\"
```

Conditional `init` Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor allowing for key bindings and variable settings to be performed as the result of tests. The following parser directives are used.

`$if`

The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test extends to the end of the line; no characters are required to isolate it.

`mode`

The `mode=` form of the `$if` directive is used to test whether Readline is in Emacs or vi mode. This may be used in conjunction with the `set keymap` command, for instance, to set bindings in the `emacs-standard` and `emacs-ctlx` keymaps only if Readline is starting out in Emacs mode.

`term`

The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the `=` is tested against both the full name of the terminal and the portion of the terminal name before the `-`. This allows `sun` to match both `sun` and `sun-cmd`, for instance.

`application`

The `application` construct is used to include application-specific settings. Each program using the Readline library sets the `application` name, and you can test for it. This could be used to bind key sequences to functions useful for a specific program. For instance, the `$if Bash` command adds a key sequence that quotes the current or previous word in Bash, as in the following example.

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\"`ef\"`"
$endif
```

`$endif`

`$endif`, as seen in the previous example, terminates an `$if` command.

`$else`

Commands in this branch of the `$if` directive are executed if the test fails.

`$include`

`$include` takes a single filename as an argument and reads commands and bindings from that file.

Sample `init` File

The following example shows an `inputrc` file that illustrates key binding, variable assignment, and conditional syntax.

```
# This file controls the behaviour of line input editing for
# programs that use the Gnu Readline library. Existing programs
# include FTP, Bash, and Gdb.
#
# You can re-read the inputrc file with C-x C-r.
# Lines beginning with '#' are comments.
#
# First, include any systemwide bindings and variable assignments from
# /etc/Inputrc
$include /etc/Inputrc

#
# Set various bindings for emacs mode.

set editing-mode emacs

$if mode=emacs

Meta-Control-h: backward-kill-word Text after the function name is
ignored

#
# Arrow keys in keypad mode
```

```

#
#\M-OD":      backward-char
#\M-OC":      forward-char
#\M-OA":      previous-history
#\M-OB":      next-history
#
# Arrow keys in ANSI mode
#
#\M-[D":      backward-char
#\M-[C":      forward-char
#\M-[A":      previous-history
#\M-[B":      next-history
#
# Arrow keys in 8 bit keypad mode
#
#\M-\C-OD":   backward-char
#\M-\C-OC":   forward-char
#\M-\C-OA":   previous-history
#\M-\C-OB":   next-history
#
# Arrow keys in 8 bit ANSI mode
#
#\M-\C-[D":   backward-char
#\M-\C-[C":   forward-char
#\M-\C-[A":   previous-history
#\M-\C-[B":   next-history
#
C-q: quoted-insert

$endif

# An old-style binding. This happens to be the default.
TAB: complete

# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word -- insert open and close double quotes
# and move to just after the open quote
"\C-x\"": "\"\C-b"
# insert a backslash (testing backslash escapes in sequences
# and macros)
"\C-x\\": "\\\"
# Quote the current or previous word
"\C-xq": "\eb"\ef\"
# Add a binding to refresh the line, which is unbound
"\C-xr": redraw-current-line
# Edit variable on current line.
#\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="

```

```
$endif

# use a visible bell if one is available
set bell-style visible

# don't strip characters to 7 bits when reading
set input-meta on

# allow iso-latin1 characters to be inserted rather than converted to
# prefix-meta sequences
set convert-meta off

# display characters with the eighth bit set directly rather than
# as meta-prefixed characters

set output-meta on

# if there are more than 150 possible completions for a word, ask the
# user if he wants to see all of them
set completion-query-items 150

# For FTP
$if Ftp
\C-xg": "get \M-?"
\C-xt": "put \M-?"
\M-": yank-last-arg
$endif
```

Bindable Readline Commands

This following documentation describes Readline commands that may be bound to key sequences.

- “Commands for Moving around in Readline”(on this page)
- “Commands for Changing Text in Readline” on page 232
- “Killing and Yanking” on page 233
- “Specifying Numeric Arguments” on page 234
- “Letting Readline Type for You” on page 235
- “Keyboard Macros” on page 235
- “Some Miscellaneous Readline Commands” on page 235

Commands for Moving around in Readline

The following documentation contains descriptions of the Readline command name, its default keybinding, and short descriptions of what commands do.

`beginning-of-line` (**C-A**)

Move to the start of the current line.

`end-of-line` (**C-E**)

Move to the end of the line.

`forward-char` (**C-F**)

Move forward a character.

`backward-char` (**C-B**)

Move back a character.

`forward-word` (**M-F**)

Move forward to the end of the next word.

`backward-word` (**M-B**)

Move back to the start of this, or the previous, word.

`clear-screen` (**C-L**)

Clear the screen leaving the current line at the top of the screen.

`redraw-current-line` (no default key binding)

Refresh the current line. By default, this is unbound.

Commands for Manipulating History with Readline

The following paragraphs describe the history commands for Readline.

`accept-line` (**Newline**, **Return**)

Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history` (**C-P**)

Move up through the history list.

`next-history` (**C-N**)

Move down through the history list.

`beginning-of-history` (**M-<**)

Move to the first line in the history.

`end-of-history` (**M->**)

Move to the end of the input history, i.e., the line you are entering.

`reverse-search-history` (**C-R**)

Search backward starting at the current line and moving up through the history as necessary. This is an incremental search.

`forward-search-history` (**C-S**)

Search forward starting at the current line and moving down through the the history as necessary.

`non-incremental-reverse-search-history` (**M-p**)

Search backward starting at the current line and moving up through the history as

- necessary using a non-incremental search for a string supplied by the user.
- `non-incremental-forward-search-history` (**M-n**)
Search forward starting at the current line and moving down through the the history as necessary using a non-incremental search for a string supplied by the user.
- `history-search-forward` (no default key binding)
Search forward through the history for the string of characters between the start of the current line and the current cursor position (the point). This is a non-incremental search. By default, this command is unbound.
- `history-search-backward` (no default key binding)
Search backward through the history for the string of characters between the start of the current line and the point.
This is a non-incremental search. By default, this command is unbound.
- `yank-nth-arg` (**M-C-y**)
Insert the first argument to the previous command (usually the second word on the previous line). With an argument, *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command.
- `yank-last-arg` (**M-., M-_)**
Insert last argument to the previous command (the last word of the previous history entry). With an argument, behave exactly like `yank-nth-arg`. Successive calls to `yank-last-arg` move back through the history list, inserting the last argument of each line in turn.

Commands for Changing Text in Readline

The following paragraphs describe commands for changing text in Readline.

- `delete-char` (**C-D**)
Delete the character under the cursor. If the cursor is at the beginning of the line, and there are no characters in the line, and the last character typed was not **C-D**, then returns EOF (end of file).
- `backward-delete-char` (Rubout)
Delete the character behind the cursor. A numeric argument says to kill the characters instead of deleting them.
- `quoted-insert` (**C-Q**, **C-V**)
Add the next character that you type to the line verbatim. This is how to insert things like **C-Q** for example.
- `tab-insert` (**M-Tab**)
Insert a tab character.
- `self-insert` (a, b, A, 1, !, ...)
Insert yourself.

`transpose-chars` (**C-T**)

Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative arguments don't work.

`transpose-words` (**M-T**)

Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word` (**M-U**)

Uppercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

`downcase-word` (**M-L**)

Lowercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

`capitalize-word` (**M-C**)

Uppercase the first letter in the current (or following) word. With a negative argument, do the previous word, but do not move point.

Killing and Yanking

The following paragraphs describe killing and yanking text.

`kill-line` (**C-K**)

Kill the text from the current cursor position to the end of the line.

`backward-kill-line` (no default key binding)

Kill backward to the beginning of the line. This is normally unbound.

`kill-word` (**M-D**)

Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`backward-kill-word` (**M-Delete**)

Kill the word behind the cursor.

`unix-line-discard` (**C-U**)

Kill the whole line the way **C-U** used to in UNIX line input. The killed text is saved on the kill-ring.

`kill-whole-line` (no default key binding)

Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word` (**M-d**)

Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word` (**M-Del**)

Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout` (**C-w**)

Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space` (no default key binding)

Delete all spaces and tabs around point. By default, this is unbound.

`kill-region` (no default key binding)

Kill the text between the point and the mark (saved cursor position). This text is referred to as the region. By default, this command is unbound.

`copy-region-as-kill` (no default key binding)

Copy the text in the region to the kill buffer, so it can be yanked right away. By default, this command is unbound.

`copy-backward-word` (no default key binding)

Copy the word before point to the kill buffer. The word boundaries are the same as `backward-word`. By default, this command is unbound.

`copy-forward-word` (no default key binding)

Copy the word following point to the kill buffer. The word boundaries are the same as `forward-word`. By default, this command is unbound.

`yank` (**C-Y**)

Yank the top of the kill ring into the buffer at point.

`yank-pop` (**M-Y**)

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `yank` or `yank-pop`.

Specifying Numeric Arguments

The following descriptions are the numeric arguments for Readline.

`digit-argument` (**M-0**, **M-1**, ... **M--**)

Add this digit to the argument already accumulating, or start a new argument. **M--** starts a negative argument.

`universal-argument` (no default key binding)

This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing `universal-argument` again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on. By default, this is not bound to a key.

Letting Readline Type for You

The following documentation details automatic Readline completions.

`complete` (**Tab**)

Attempt to do completion on the text before point. This is implementation defined. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion.

`possible-completions` (**M-?**)

List the possible completions of the text before point.

`insert-completions` (**M-***)

Insert all completions of the text before point that would have been generated by `possible-completions`.

`menu-complete` (no default key binding)

Similar to `complete`, but replaces the word to be completed with a single match from the list of possible completions.

Repeated execution of `menu-complete` steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung and the original text is restored. An argument of n moves n positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to **Tab**, but is unbound by default.

Keyboard Macros

The following descriptions are for keyboard macros.

`start-kbd-macro` (**C-x** **(**)

Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro` (**C-x** **)**)

Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro` (**C-x** **e**)

Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Some Miscellaneous Readline Commands

The following documentation details some miscellaneous Readline commands.

`reread-init-file` (**C-X**, **C-R**)

Read in the contents of your `~/.inputrc` file, and incorporate any bindings or variable assignments found there.

abort (C-G)

Stop running the current editing command and ring the terminal's bell (subject to the setting of `bell-style`).

do-uppercase-version (M-a, M-b, M-x, ...)

If the metafiled character, **x**, is lowercase, run the command that is bound to the corresponding uppercase character.

÷_prefix-meta (Esc)

Make the next character that you type be metafiled. This is for people without a meta key. Using the keystroke sequence, **Esc f**, is equivalent to using **M-f**.

undo (C-_)

Incremental undo, separately remembered for each line.

revert-line (M-R)

Undo all changes made to this line. This is like typing the `undo` command enough times to get back to the beginning.

tilde-expand (M-~)

Perform tilde expansion on the current word.

set-mark (C-@)

Set the mark to the current point. If a numeric argument is supplied, the mark is set to that position.

exchange-point-and-mark (C-x C-x)

Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

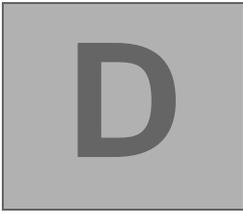
character-search (C-])

A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.

Readline in vi Mode

While the Readline library does not have a full set of vi editing functions, it does contain enough to allow simple editing of the line.

In order to switch interactively between Emacs and vi editing modes, use the command **M-C-J** (`toggle-editing-mode`). When you enter a line in vi mode, you are already placed in `insertion` mode, as if you had used an **i** keystroke. Using **Esc** switches you into `edit` mode, where you can edit the text of the line with the standard vi movement keys, move to previous history lines with **k**, the following lines with **j** (and so forth).



Using History Interactively

The GNU History Library provides a history expansion feature similar to the history expansion in `csh`. History expansion takes two parts: determining which line from the previous history will be used for substitution, called the *event* (see “Event Designators” on page 238), and selecting portions of that line for inclusion into the current line, called *words* (see “Word Designators” on page 238). GDB breaks the line into words in the same way that the `bash` shell does, so that several English (or UNIX) words surrounded by quotes are considered one word.

The following documentation describes how to use the GNU History Library interactively.

- “Event Designators” on page 238
- “Word Designators” on page 238
- “Modifiers” on page 239

Event Designators

An *event designator* is a reference to a command line entry in the history list.

- !
Start a history substitution, except when followed by a space, tab, or the end of the line... = or (.
- !!
Refer to the previous command. This is a synonym for !-1.
- !n
Refer to command line *n*.
- !-n
Refer to the command line *n* lines back.
- !string
Refer to the most recent command starting with *string*.
- !?string[?]
Refer to the most recent command containing *string*.

Word Designators

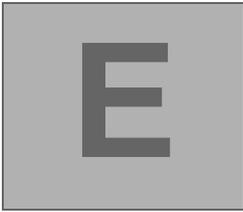
A **:** separates the event designator from the *word designator*. It can be omitted if the word designator begins with any of the `^`, `$`, `*` or `%` characters. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

- 0 (zero)
The zero'th word. For many applications, this is the command word.
- n
The *n*'th word.
- ^
The first argument; that is, word 1.
- \$
The last argument.
- %
The word matched by the most recent `?string?` search.
- x-y
A range of words; `-y` abbreviates `0-y`.
- *
All of the words, excepting the zero'th. This is a synonym for `1-$`. It is not an error to use `*` if there is just one word in the event; the empty string is returned in that case.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`.

- # The entire command line typed so far. This means the current command, not the previous command.
- h Remove a trailing pathname component, leaving only the head.
- r Remove a trailing suffix of the form `. suffix`, leaving the basename.
- e Remove all but the suffix.
- t Remove all leading pathname components, leaving the tail.
- P Print the new command but do not execute it.



Formatting Documentation

The already-formatted reference card is available, ready for printing with PostScript or Ghostscript, in the GDB subdirectory of the main source directory. If you can use PostScript or Ghostscript with your printer, you can print the reference card immediately with the `refcard.ps` file. You can format the file, using TEX, by using the `make refcard.dvi` command.

The GDB reference card is designed to print in landscape mode on US letter size paper (a sheet 11 inches in width and 8.5 inches in length). You will need to specify this form of printing as an option to your DVI output program. All the documentation for GDB comes as part of the machine-readable distribution. The documentation is written in Texinfo format, which is a documentation system that uses a single source file to produce both online information and a printed manual. You can use one of the Info formatting commands to create the online version of the documentation and TEX (or `texi2roff`) to typeset the printed version. GDB includes an already formatted copy of the online Info version of this manual in the `gdb` subdirectory. The main Info file is `gdb.info`, and it refers to subordinate files matching `gdb.info*` in the same directory. If necessary, you can print out these files, or read them with any editor; but they are easier to read using the `info` subsystem in GNU Emacs or the standalone `info` program, available as part of the GNU Texinfo distribution. If you want to format these Info files yourself, you need one of the Info formatting programs, such as `texinfo-format-buffer` or `makeinfo`.

If you have `makeinfo` installed, and are in the top level GDB source directory, you can

make the Info file by typing:

```
cd gdb
make gdb.info
```

If you want to typeset and print copies of this manual, you need TEX, a program to print its DVI output files, and `texinfo.tex`, the Texinfo definitions file. TEX is a typesetting program; it does not print files directly, but produces output files called dvi files. To print a typeset document, you need a program to print dvi files. If your system has TEX installed, chances are it has such a program. The precise command to use depends on your system; `lpr -d` is common; another (for PostScript devices) is `dvips`. The DVI print command may require a file name without any extension or a `.dvi` extension. TEX also requires a macro definitions file called (`texinfo.tex`). This file tells TEX how to typeset a document written in Texinfo format. On its own, TEX cannot either read or typeset a Texinfo file. `texinfo.tex` is distributed with GDB and is located in the `gdb-version-number/texinfo` directory. If you have TEX and a DVI printer program installed, you can typeset and print this manual. Change to the `gdb` subdirectory of the main source directory and then use the `make gdb.dvi` command.

Index

Symbols

- ! (NOT operator) 159
- #
 - for comments 165
 - Modula-2 inequality operator 113
 - prompt 147
- \$ (value in history) 116
- & (bitwise AND) for C and C++ 102
- , (sequencing operator)
 - for C, C++ 101
 - for Modula-2 107
- , in options 24
- , in options 24
- .o file 133
- :: (double-colon)
 - C++ (scope resolution operator) 103
 - Modula-2 (scope operator) 108
 - specifying a variable 78
- := (for assignment) for Modula-2 107
- = (for assignment) for C and C++ 102
- ? key, for help 31
- ?: (ternary operator) for C and C++ 102
- @ (array operator) 78, 102, 106, 108
- [] (array indexing operator) 103
- \ (backslash), for escape sequences 167
- \n (newline escape) 166
- | (bitwise OR) for C and C++ 102
- || (logical OR) for C and C++ 102
- ^ (bitwise exclusive OR) for C and C++ 102
- ~ 103

A

a.out 14

- address, locating 81
- add-shared-symbol-file 128
- add-symbol-file 128
- aliases 20
- AMD 133, 145
- arguments 37
- array 80, 85, 104–105
- Array Tech LSI33K RAID controller board 134
- assembler source file 97
- attach 40, 132
- automatic display 83
- awatch 50

B

- b 151
- BACKSPACE key 30
- backtrace 67, 173
- backtrace 67
- batch 26
- batch mode 26
- BFD 10–11, 18–19, 132
- boolean types 107
- break 47, 62
- breakpoint 42, 46
 - command (breakpoint, or b) 137
 - condition 54
 - defining functions with C++ 106
 - deleting 52
 - displaying 191
 - enabling, disabling 192
 - menus 57
 - with Insight, setting 203
- bugs, reporting 215
- buttons, Insight 172–176

C

C++

- exception handling 106

C, C++

- & (bitwise AND) 102
- , (sequencing operator) 101
- = (for assignment) 102
- ?: (ternary operator) 102
- | (bitwise OR) 102
- || (logical OR) 102
- ^ (bitwise exclusive OR) 102

- compatibility 101
- constants 104
- operators 101–103
- source file 96

- call 122

- call stack 65, 182

- catch 51

- catch catch 106

- catch throw 106

- catchpoint 46, 51

- character constants 104

- character types 107

- checksum 140

- child process 35, 41

- CHILL source file 97

- clear 53

- COFF 14

- command 25

- command history file 158

- compiling 36

- condition 55

- configure 18, 20

- confirmation requests 162

- connect 148

- constants 103–104

- contacting Red Hat iii

- continue 58, 63, 213

- continuing 58

- convenience variables 90

- core 25

- core dump file 25, 125, 133

- core-file 127, 132

- CPU simulator 155

- CPU time 42

D

- data spaces 122

- data type 81–82

- debugger

- defined 5

- GUI 171

- debugging

- call stack 182

- editor, aborting 186

- functions, selecting 180

- remote 10, 135, 141

- remote serial protocol 136

- source code settings with Insight 173

- source files, selecting 180

- stack frame 182

- stub, using 136

- stubs 141

- symbol file errors 129

- target, specifying 131

- define 163–165

- delete 53

- delete display 84

- designators 238

- detach 40, 147

- directories, specifying 74

- directory 25, 74

- disable 54

- disable display 84

- disassemble 75, 213

- disassembly 10

- display 84

- document 164

- down 68, 213

- dump file 125

- DWARF 16

- dynamic arrays, debugging, with Modula-2 112

- dynamic linking 127

E

- echo 166

- ECOFF 15

- editor

- aborting, with Insight 186

- alternates (or external) 179

- ELF 15

- Emacs 158, 211–214, 226–227

- buffer 211–212

- escape sequences 226

- shell mode 212

- enable 54

- enable display 84

- end 56

- enumerated constants 104

- environment 37–39

- EPROM/ROM code debugging 48

- errors 119, 129, 215

- ESC, and the ? (question mark) key 31

- EST-300 ICE monitor 134

- event designators 238

- examining memory 82

- exception handling 52, 69, 106, 138

- exceptionHandler 139

- exec 25

- exec-file 126, 132

- expression, regular 73

- expressions 78, 92, 104, 187

- expressions, regular 106

F

- f 26
- file 126, 150
- file, specifying 125–129
- finish 59, 63, 122, 213
- floating point
 - constants 104
 - hardware 93
 - registers 91
- floatingpoint
 - types 107
- flush_i_cache 139
- fork 44
- frame 66, 68–69
- frame pointer register 66
- frame stack 182
- frames 66
- Fujitsu SPARClite boards 134
- fullname 26
- functions 11, 115

G**GDB**

- # (comment) 165
- absolute file names, converting 129
- address ranges 41
- altering execution 119
- arguments 38
- array 80
- backtrace 173
- batch mode 26
- BFD 10–11, 16, 18–19
- breakpoint 46
- building 18
- C 101
- C++ 101
- C, C++
 - constants 103
 - operators 101–103
- catchpoint 46, 51
- changing to a different file 125
- checksum 140
- child process 35
- choosing files 23
- choosing modes 23
- command 238
 - completion 29–30, 33, 107
 - file 163, 165
 - history 158
 - options 24
 - repeating 29
 - syntax 29
 - truncated 29
 - using 140
- compiling 36
- complete 33

- condition 55
- configuring 19
- continuing 58
- contributors 7–10
- copying 33
- core 25
- core dump 24–25
- core dump file 41, 125
- data spaces 122
- debugging, remote 135
- delimiters 115
- disassembly 10
- E7000 153
- EBMON protocol 145
- Emacs 158, 211–214, 226–227
- environment 37, 39
- environment variables, setting expressions 33
- errors 119, 129, 215
- executable files, core files 24
- exiting 27
- expressions 10, 33, 79, 104, 119
- file, specifying 125, 129
- filter 26
- frame 65
- gdbserve.nlm 143
- gdbserver 142
- GUI 171
- Hitachi 152
- hooks 165
- host 10, 18, 20–21
- info 33
- init files 165
- input and output 37, 39
- installation 18
- instruction scheduling 36
- Intel 960, using Nindy 144
- interrupt signal 121
- interrupting 27
- invoking 23–24, 36
- key bindings 223, 225
- kill 41
- language
 - setting 105
 - specific information 95
- list 30, 212
- make 18
- memory
 - allocation 106
 - arrays 78
 - mapping 18, 25
 - values 119
- MIPS 153
- Modula-2
 - deviations 111
 - functions 109
 - operators 107–108
 - sets 109
 - variables 109
- numbering 160

- object file formats 14
- opcode tables 18
- operators
 - C, C++ 78
 - Modula-2 109
- optimizing 36
- output 37, 166
- output formats 81
- overloading 57, 106
- parentheses 31
- path searches 38
- printf 167
- process 45
 - information 41
 - purposes 45
 - stopped 42
 - stopping 138
- program information 33
- prompt 158
- protocol 132, 140
- ps utility 40
- quiet mode 26
- quitting 23
- quotes 31
- range checking 98, 105
- readline init file 223
- readline interface 158
- read-only files 122
- registers 10
- regular expression 48
- remote debugging 10, 27, 141
- remote serial protocol 135–136, 142–143
- requirements 7
- restarting 119
- scheduler 63
- scratch area 122
- screen size, manipulating 160
- search 73, 212
- searches 38
- set 33
- sh 18
- shared libraries 129
- shell 18
- shell behavior 237
- shell commands 23, 27
- show 33, 160
- signal 60, 121
- signals 61
- simulator 155
- SPARCllet, connecting to 151
- stabs 15, 104
- stack 173
- stack frame 10–11, 26, 105, 122
- start-up commands 125
- state, showing 33
- stepping 58
- stopping a process 23
- stubs 141
- subprocess 26

- symbol 129
- symbol files 11
- symbol table 10, 12, 25, 89, 104, 107, 115, 125, 142
- target remote 141
- target, defined 10
- targets, specifying 131
- TCP connection 142
- terminal modes 39
- thread 42–43, 51, 63
- type
 - checks
 - Modula-2 112
 - type and range checking 105
 - type and range checks
 - C, C++ 105
 - Modula-2 112
 - type checking 98
 - variables 10, 105, 119
- version 33
- vi 224
- VxWorks 148
- warnings 129, 161
- warranty 33
- watchpoint 46, 50
- working directory 37, 39
- Z8000 155
- GDBrun 36
- gdbserve.nlm 143
- gdbserver 141
- getDebugChar 139
- global variables 178

H

- handle 61
- handle_exception 137
- hbreak 48
- help 20
- help 24, 32, 151, 164
- help target 132
- history
 - references 90
 - showing 237
 - symbol table 89
- history numbers 89
- Hitachi 133
- hooks 165
- host 10, 20–21
- HPPA, Winbond 134

I

- IDP board 134
- if 164
- ignore 55
- include 18

- info 129
- info address 115
- info args 69
- info breakpoints 49
- info catch 69
- info display 84
- info files 132
- info frame 69, 98
- info functions 117
- info line 75
- info locals 69
- info registers 91
- info signals 61
- info source 98, 116
- info sources 117
- info target 132
- info types 116
- info variables 117
- info watchpoints 49, 51
- init files 165
- Insight 171–207
 - assembly code, displaying 180
 - backtrace 173
 - breakpoints
 - appearance 177–178
 - information balloon 178, 207
 - setting 177, 202–204
 - buttons 171–176
 - Add Watch 187
 - Run 172
 - Stop 172
 - color dialog box 173
 - convenience variables 187
 - display panes 173
 - drop-down lists 180
 - editing, aborting (with Escape key) 186
 - expression 178, 186–187
 - file drop-down list menu 179
 - File menu 172
 - function
 - drop-down combo box 179, 196
 - drop-down list box 180
 - Function Browser, with source browser 196
 - functions
 - selecting 180
 - Global Preferences, setting 174
 - HTML help 198
 - icons 174
 - information balloon 178
 - jumps 181, 201
 - local variables 178, 202
 - Local Variables, Variable menu 188
 - menus 171
 - File (Source Window) 172
 - Open (Load New Executable dialog) 172
 - Source Window 172
 - Watch (Watch Expressions window) 186
 - mouse, using 176
 - Open menu 172

- pointers, casting 187
- preferences, settings to use 173
- registers, debugging 183
- scroll bar, using 179, 196
- search 200
- selecting source files to debug 180
- source
 - code, displaying 173, 180
 - file, debugging 200
 - preferences, settings to use 173
 - selecting files to debug 180
- stack frame 182
- starting 171
- status text box 179, 196
- tutorials 199
- variables 178, 204
- Watch menu 186
- windows
 - Breakpoints 191
 - Function Browser 195–196
 - Help 198
 - Local Variables 188
 - Memory 184
 - Registers 183
 - Source Window 172–181
 - Watch Expressions 186
- int getDebugChar 137
- integer constants 103
- integral types 107
- Intel 960 134

J

- jump 120

K

- key bindings 223
- kill 41, 121
- killing text, defined 221

L

- language
 - displaying source 97
 - setting 96, 105
- libiberty 18
- linking, dynamic 127
- list 72, 212–213
- load 51, 127
- local variables 69, 79, 188, 204

M

- machine registers 91
- maint info breakpoints 49

- maint print symbols 117
- make 18, 28
- member function calls 104
- memory 106
 - arrays 78
 - examining 82
 - mmap 25
 - preferences, setting 184
 - symbols 25
- memset 138–139
- META key 31
- mmalloc 18
- modifiers 239
- Modula-2 95, 107
 - , (sequencing operator) 107
 - := (for assignment) 107
 - constants 110
 - defaults 111
 - deviations 111
 - extensions 96
 - functions 109
 - procedures 109
 - scope 112
 - sets 109
 - type and range checks 112
 - type checking 99
 - variables 109, 112
- Modula-2 operators 107–108
- Motorola 68000 134
- multiple threads 42

N

- Netware Loadable Modules 15
- newline 166
- next 59, 63, 213
- nexti 60, 213
- Nindy Monitor 134
- nlnmread.c 15
- numbering convention 160
- nx 26

O

- object file formats 14–15
- OKI HPPA board 134
- opcodes 18
- operators 78, 99
 - C, C++ 101–103
 - Modula-2 107–109
- output
 - formats 81
 - suppressing 166
- output 167
- overloading 57, 106

P

- path 38
- pc (program counter) 176
- PE 15
- pointer constants 104
- pointer types 107
- preprocessor commands 78
- print 77, 115
- print settings 85
- printf 167
- problems, reporting iii
- processes 35, 42
- protocol, remote serial 135
- protocols for targets 132
- ptype 106, 116
- putDebugChar 139

Q

- quiet 26
- quit 27, 160

R

- range checking 100
- range checks
 - C, C++ 105
 - Modula-2 112
- rbreak 48, 106
- Readline 220
- readline 18
- readline key bindings 225
- readnow 25
- registers 10, 91–92, 187
 - debudding 183
 - relativized value 91
 - stack 93
- regular expression 73
- regular expressions 106
- remote debugging 10, 135, 141
- remote serial protocol 135–136, 143
- return 122
- reverse-search 73
- run 151–152
- running process 40
- rwatch 50

S

- scalar types 107
- scheduler 63
- scope 105
- scope resolution 79
- scratch area 122
- se 25
- search (command) 73, 212

section 128
 select-frame 66
 serial protocol 135
 session ID 42
 set 38, 120, 157–158
 set assembly-language 76
 set check range 100
 set check type 99
 set complaints 161
 set confirm 162
 set demangle-style 88
 set editing 158
 set endian 134
 set gnutarget 132
 set height 160
 set heuristic-fence-post 70
 set history 158
 set input-radix 160
 set language 96
 set memory 153
 set output-radix 161
 set print 106
 set print address 85
 set print array 86
 set print elements 87
 set print max-symbolic-offset 86
 set print symbol-filename 86
 set print union 106
 set processor 154
 set prompt 158
 set remotedebug 154
 set retransmit-timeout 154
 set rstack_high_address 93
 set scheduler-locking mode 63
 set symbol-reloading 117
 set timeout 154
 set types 107
 set verbose 161
 set width 160
 set write 123
 set_debug_traps 137
 sh 18
 share 129
 sharedlibrary 129
 shell 27
 SHELL environment variable 39
 show 38, 160
 show commands 159
 show complaints 161
 show convenience 91
 show demangle-style 88
 show directories 74
 show editing 158
 show gnutarget 133
 show height 160
 show heuristic-fence-post 70
 show history 159
 show input-radix 161
 show language 98
 show memory 153
 show output-radix 161
 show print 106
 show print address 85
 show print array 86
 show print elements 87
 show print max-symbolic-offset 86
 show print symbol-filename 86
 show print union 106
 show processor 154
 show prompt 158
 show range 100
 show remotedebug 154
 show retransmit-timeout 154
 show scheduler-locking 63
 show symbol-reloading 117
 show timeout 154
 show user 164
 show values 90
 show verbose 161
 show width 160
 show write 123
 signal 61
 program 60
 signal (command) 121
 simulator 133, 155
 single-stepping 62
 SOM 15
 source
 filename 166
 files 71
 path 74
 stabs 15, 104
 stabs 101
 stack
 backtrace 173
 Insight, using 182
 stack frame 10–11, 26, 105, 122
 defined 65
 step 58, 63, 151, 213
 stepi 60, 213
 stepping 58
 string constants 104
 structures 85
 stubs 135–136, 141
 sub-routines 136
 symbol 11, 85, 106, 115, 118
 definitions 129
 filenames 86
 table 115
 symbol 25
 symbol files 11
 symbol table 10, 12, 25, 104, 107, 115
 symbol-file 118, 126

T

TAB key 30

- Tandem ST2000 134
- target 132–133, 145, 147, 153
- target core 133
- target exec 133
- target remote 133, 139, 141
- target sim 155
- target, defined 10
- targets 20–21
 - classes 131
 - core files 131
 - defined 131
 - executables 131
 - processes 131
- tbreak 48, 54
- tcatch 52
- Texinfo 241
- thbreak 48
- thread 43, 62
- threads 42, 63, 197
- timeout 150
- tty 27
- type 11, 115
 - C, C++ checks 105
 - Modula-2 107
 - Modula-2 checks 112
 - range checking 105

U

- UDI (Universal Debugger Interface) protocol 133, 145
- undisplay 84
- union 106
- unload 51
- unset environment 39
- until 59, 63
- up 68

V

- value
 - history 81, 89
- variables 11, 78, 105, 115, 178
 - convenience 187
 - in expressions 78
- vi 224
- virtual function table 89, 106
- void exceptionHandler 138
- void flush_i_cache 138
- void putDebugChar 138
- VxWorks 134, 148

W

- W89K monitor 134
- watch 50
- watchpoint
 - defined 46
 - multi-thread programs 51
 - setting 50
- whatis 116
- while 164
- width, with set 120
- windows, Insight 172
- word designators, defined 238

X

- x command 82
- XCOFF 15

Y

- yanking, defined 221