

Total/db User's Guide

LynxOS Release 4.0
DOC-0409-00

Product names mentioned in *Total/db User's Guide* are trademarks of their respective manufacturers and are used here for identification purposes only.

Copyright ©1987 - 2002, LynuxWorks, Inc. All rights reserved.
U.S. Patents 5,469,571; 5,594,903

Printed in the United States of America.

All rights reserved. No part of Total/db User's Guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photographic, magnetic, or otherwise, without the prior written permission of LynuxWorks, Inc.

LynuxWorks, Inc. makes no representations, express or implied, with respect to this documentation or the software it describes, including (with no limitation) any implied warranties of utility or fitness for any particular purpose; all such warranties are expressly disclaimed. Neither LynuxWorks, Inc., nor its distributors, nor its dealers shall be liable for any indirect, incidental, or consequential damages under any circumstances.

(The exclusion of implied warranties may not apply in all cases under some statutes, and thus the above exclusion may not apply. This warranty provides the purchaser with specific legal rights. There may be other purchaser rights which vary from state to state within the United States of America.)

Contents

PREFACE	VII
For More Information	vii
Typographical Conventions	viii
Special Notes	ix
Technical Support	ix
LinuxWorks U.S. Headquarters	ix
LinuxWorks Europe	ix
World Wide Web	ix
CHAPTER 1	TOTAL/DB OVERVIEW	1
	Included Components	1
	GDB	1
	GDBServer	1
	Insight	2
	SSPP	2
	SKDB	2
	User Process Debugging vs. Kernel/Device Driver Debugging	2
	Local Debugging versus Remote Debugging	3
	Local Debugging	4
	Remote Debugging	4
	Total/db Configuration Options	7
	Supported Languages	8
	Source Code	9
CHAPTER 2	DEBUGGING WITH GDB	11
	The GNU Source Level Debugger	11
	GDB as Free Software	11

Controlling GDB	12
Prompt	12
Command Editing	12
Command History	13
Screen Size	14
Numbers	15
Optional Warnings and Messages	16
Getting In and Out of GDB	17
Invoking GDB	17
Quitting GDB	21
Shell Commands	22
GDB Commands	22
Command Syntax	22
Command Completion	23
Getting Help	25
Running Programs under GDB	27
Compiling for Debugging	27
Starting Your Program	28
Your Program's Arguments	30
Your Program's Environment	30
Your Program's Working Directory	31
Your Program's Input and Output	32
Debugging an Already-Running Process	32
Killing the Child Process	33
Debugging Programs with Multiple Threads	34
Debugging Programs with Multiple Processes	35
Stopping and Continuing	36
Breakpoints, Watchpoints, and Exceptions	36
Continuing and Stepping	50
Signals	53
Stopping and Starting Multithread Programs	55
Examining the Stack	56
Stack Frames	57
Backtraces	58
Selecting a Frame	59
Information about a Frame	60
MIPS Machines and the Function Stack	61
Examining Source Files	62
Printing Source Lines	62
Searching Source Files	65

Specifying Source Directories	65
Source and Machine Code	66
Examining Data	67
Expressions	68
Program Variables	69
Artificial Arrays	70
Output Formats	71
Examining Memory	72
Automatic Display	74
Print Settings	76
Value History	82
Convenience Variables	83
Registers	84
Floating Point Hardware	86
Using GDB with Different Languages	86
Switching between Source Languages	87
Displaying the Language	89
Type and Range Checking	89
Supported Languages	92
Examining the Symbol Table	99
Altering Execution	102
Assignment to Variables	103
Continuing at a Different Address	103
Giving Your Program a Signal	104
Returning from a Function	105
Calling Program Functions	105
Patching Programs	106
GDB Files	106
Commands to Specify Files	106
Errors Reading Symbol Files	110
Specifying a Debugging Target	112
Active Targets	112
Commands for Managing Targets	113
Remote Debugging	114
Stored Command Sequences	116
User-Defined Commands	116
User-Defined Command Hooks	118
Command Files	118
Commands for Controlled Output	119

Using GDB under GNU Emacs	120
Command Line Editing	124
Introduction to Line Editing	124
Readline Interaction	124
Readline Init File	127
Using History Interactively	134
History Interaction	135

CHAPTER 3 LYNXOS GDB ENHANCEMENTS 137

Overview	137
Debugging POSIX Threads	138
Understanding Thread Numbers	138
Browsing and Switching Threads	138
Setting a Breakpoint	139
Resuming Threads	140
Debugging Embedded Applications Remotely	140
Using the Target Command	141
Debugging Remote Targets	142
Supported Protocols for Remote and Extended-Remote Targets	142
Starting the Remote Target	145
Target's Environment	148
Postmortem Debugging of Dynamically Linked Programs	148
Debugging Shared Libraries	148
Creating a Shared Library for Debugging Purposes	149
Loading Shared Library Symbol Information	149
Deferred Breakpoints	150
Shared Library File Path Names	153
Symbol Table	157
Single-Stepping into a Shared Library Function	157
Summary of Additional Commands for Shared Library Support	158
Debugging Kernel/Device Drivers	158
Requirements	158
Building a Kernel for Debug Purposes	159
Debugging the Kernel	159
Loading Device Drivers Dynamically	163
Raw SKDB Commands	164
Proxy Server	165
Syntax	166
Installation	166

General Tips and Miscellaneous Issues	168
Reading and Writing Large Memory Blocks	168
Browsing Target Process's Environment	169
Executing Remote Shell Commands	169
Function Calls in a Multithreaded Process	170
Functions Calls after Ctrl+C	171
Resuming after a Blocking System Call	171
Debugging a Signal-Intensive Process	172
CHAPTER 4	DEBUGGING WITH TOTAL/DB
	173
Source Window	174
Toolbar Buttons	180
Special Display Pane Features	182
Using the Mouse in the Display Pane	182
Below the Horizontal Scroll bar	186
Dialog boxes for the Source Window	189
Stack Window	197
Registers Window	198
Memory Window	200
Memory Preferences Dialog Box	201
Watch Expressions Window	202
Add Watch Button	204
Watching Registers	204
Casting Pointers in the Watch Expressions Window	204
Local Variables Window	205
Breakpoints Window	206
Console Window	209
The Function Browser Window	210
Help Window	213
Tutorials for Debugging with Insight	215
Initializing a Target Executable File	215
Console Window with Initial Commands	216
Setting Breakpoints and Viewing Local Variables	218
CHAPTER 5	SIMPLE KERNEL DEBUGGER - SKDB
	223
Overview	223
Installing/Removing SKDB	224
Installing SKDB	224

Removing SKDB	224
Using SKDB	224
SKDB Prompt	224
Starting SKDB Automatically after a Kernel Crash or Panic	225
Breaking into SKDB with Hot Key	225
Kernel Status Display	226
Kernel Status Redisplay	227
Stack Trace Display	227
Verbose Trace Mode	227
Process, Thread, and Other Displays	227
Resuming the Kernel	227
Setting Breakpoints	228
Single-Stepping	228
Disassembly	228
Setting Watchpoints	229
SKDB Commands	231
General Notes	234
Parameter Validation	234
Symbol Information	234
Address Expressions	234
Default Virtual Address Space	235
Remote Debugger Interface Protocol	235

APPENDIX A	GNU SOFTWARE LICENSE AGREEMENT	237
	GNU General Public License	237
	Preamble	237
	Terms & Conditions for Copying, Distribution and Modification	238
	How to Apply these Terms to Your New Programs	243
	Contributors to GNU CC	245
	Protect Your Freedom; Fight “Look And Feel”	248

INDEX	253
--------------------	------------

— Preface

This Total/db guide contains information about debugging LynxOS targets with the Total/db debugger. This manual assumes that you have a basic understanding of debugging high-level language program and is intended primarily for developers of LynxOS. A few tasks in this manual may require `root` privileges on the host system or other information that typically falls in the system administration domain.

For More Information

For more information on the features of LynxOS, refer to the following printed and online documentation.

- *LynxOS Release Notes*

This printed document contains late-breaking information about the current release.

- *LynxOS Installation Guide*

This manual supports the initial installation and configuration of LynxOS and the X Windows System.

- *LynxOS User's Guide*

This document contains information about basic system administration and kernel level specifics of LynxOS. It contains a “Quick Starting” chapter and covers a range of topics, including tuning system performance and creating kernel images for embedded applications.

- Online information

Information about commands and utilities is provided online in text format through the `man` command. For example, a user wanting

information about the GNU compiler would use the following syntax, where `gcc` is the argument for information about the GNU compiler:

`man gcc`

More recent versions of the documentation listed here may also be found online.

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to file names and commands are case sensitive and should be typed accurately.

Kind of Text	Examples
Body text; <i>italicized</i> for emphasis, new terms, and book titles	Refer to the <i>LynxOS User's Guide</i> .
Environment variables, file names, functions, methods, options, parameter names, path names, commands, and computer data	<code>ls</code> <code>-l</code> <code>myprog.c</code> <code>/dev/null</code>
Commands that need to be highlighted within body text, or commands that must be typed as is by the user are bolded .	login: myname # cd /usr/home
Text that represents a variable, such as a file name or a value that must be entered by the user	<code>cat filename</code> <code>mv file1 file2</code>
Blocks of text that appear on the display screen after entering instructions or commands	<pre> Loading file /tftboot/shell.kdi into 0x4000 File loaded. Size is 1314816 Copyright 2000 LynuxWorks, Inc. All rights reserved. LynxOS (ppc) created Mon Jul 17 17:50:22 GMT 2000 user name: </pre>
Keyboard options, button names, and menu sequences	Enter , Ctrl-C

Special Notes

The following notations highlight any key points and cautionary notes that may appear in this manual.

NOTE: These callouts note important or useful points in the text.



CAUTION! Used for situations that present minor hazards that may interfere with or threaten equipment/performance.

Technical Support

LynuxWorks Technical Support is available Monday through Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Europe).

The LynuxWorks World Wide Web home page provides additional information about our products and LynuxWorks news groups.

LynuxWorks U.S. Headquarters

Internet: support@lnxw.com

Phone: (408) 979-3940

Fax: (408) 979-3945

LynuxWorks Europe

Internet: tech_europe@lnxw.com

Phone: (+33) 1 30 85 06 00

Fax: (+33) 1 30 85 06 06

World Wide Web

<http://www.lynuxworks.com>

Total/db is a robust and powerful debugger tool chain that supports debugging of various LynxOS targets. Its modularity allows a variety of configurations suitable for the needs of particular applications.

Included Components

Total/db consists of the following component programs. Each program runs as a separate process:

GDB

GDB is the GNU debugger and is the “core” of Total/db. LynuxWorks has improved and enhanced GDB in a variety of ways for better debugging LynxOS targets. Readers are advised to read Chapter 2, “Debugging with GDB” on page 11 to get general familiarity with GDB and then the following chapter Chapter 3, “LynxOS GDB Enhancements” on page 137 for the LynuxWorks GDB-specific issues.

GDBServer

GDBServer is a part of the GDB package. GDBServer serves as the remote debug target agent for remote user process debugging. For its details, refer to Chapter 3, “LynxOS GDB Enhancements” on page 137.

Insight

Insight is the graphical user interface (GUI) front-end for GDB. It runs under the X-Window system and provides GDB with an intuitive GUI particularly good at displaying complex data.

Insight provides only the user interface using GDB as the “debug engine.” It knows very little about the debug target, is independent from the debug targets, and is usable with a variety of different debug targets, including LynxOS applications.

SSPP

SSPP is a simple proxy server program that extends the physical reach of serial line remote debugging using GDB. Refer to Chapter 3, “LynxOS GDB Enhancements” on page 137.

SKDB

SKDB is a simple machine-level symbolic kernel debugger. SKDB provides interactive access to the LynxOS kernel internals including device drivers. It works as the debug agent for remote kernel debugging with GDB. Refer to Chapter 5, “Simple Kernel Debugger - SKDB” on page 223.

User Process Debugging vs. Kernel/Device Driver Debugging

LynxOS distinguishes CPU execution modes between user and supervisor. A user process runs in user mode with limited privileges. Supervisor mode controls all kernel activities including system calls, device drivers, interrupt handling, and so forth.

Due to the differences in exception handling and other operations between the two modes, appropriate consideration must be given before selecting a debugging tool. Table 1-1 lists the options available for each CPU execution mode.

Table 1-1: User Process Debugging vs. Kernel/Device Driver Debugging

Target \ Use	Stand-alone	Source level	Source level w/ GUI
User Process	GDB ¹	GDB	Insight+GDB
Kernel/Device Driver	SKDB	GDB+SKDB ²	Insight+GDB+SKDB

1. Same as source level
2. Remote debug only

NOTE: Total/db can debug only a single process per debug session for user process debugging. To debug multiple processes, do as many Total/db sessions as the number of target processes involved. There is no synchronization mechanism provided between those sessions.

For kernel debugging, a single Total/db session controls and debugs the entire kernel. It is possible to mix user process debugging and kernel debugging by invoking multiple Total/db sessions. Severe interference by the kernel debugging session with the user process debugging is anticipated because the entire operating system will freeze while it is at a kernel breakpoint.

Local Debugging versus Remote Debugging

LynxOS can be configured from an embedded system to a full-featured workstation with different levels of available resources including the user interface and file systems. If the target system has enough resources, it is possible to set up the debugger on the same machine; this is called *local debugging*. If the target is poor in resources or if a more powerful or different host workstation is preferred, perform *remote debugging* over a communication channel.

Local Debugging

User Process Debugging

GDB can run locally on the target machine for full source level debugging. Insight can also run on the same machine to provide GUI.

Kernel/Device Driver Debugging

SKDB provides stand-alone machine level local debugging on a serial terminal port or a video console. There is no source level or GUI kernel debugging (see Table 1-2).

Table 1-2: Local Debugging

Target \ Component	Insight	GDB	SKDB
User Process	Optional for GUI	Character-based source level debugging	N/A
Kernel/Device Driver	N/A	N/A	Character-based machine level debugging

Remote Debugging

Remote debugging uses two machines, one running the LynxOS target application is called the *debug target*, the other running the debugger is called the *debug host*. The debugger program on the debug host communicates with the *remote debug agent* program on the debug target using a remote debug protocol through the communication channel. The debug agent, the remote debug protocol, and the communication channel differ between user process debugging and kernel/device driver debugging.

Symbol Files

In remote debugging, both the debug host and the debug target must have the same compiled binary image files. The debug host uses the files for obtaining symbols and other debug information while the debug target uses the files for actual

program execution. The debug target's files may be stripped of symbols in order to reduce their size, but they must be synchronized with the host's files; otherwise, the debugger may behave incorrectly or unexpectedly.

User Process Debugging

There are two choices for the communication channel:

- TCP/IP provides reliable and fast communication, but it may not be available on simple embedded targets
- A serial line such as the RS-232 may be available on most targets for remote debugging but it is usually slower and less reliable than TCP/IP communication. A serial line also limits the distance between the host and target (see SSPP below).

With either communication channel, GDBServer must run on the target as the remote debug agent. GDBServer is a much smaller program (~100KB) than GDB and it translates the remote debug protocol into debug system calls and vice versa. Once a remote debug communication is established, there is no difference between serial line debugging and TCP/IP debugging.

One can optionally run Insight as a GUI (graphical user interface). Insight can run on the same host as the GDBs, or yet another host machine, in which case Insight communicates with GDB through a TCP/IP channel. Because Insight uses the X-Window, the display server (user interface) can be run on another machine.

Kernel/Device Driver Debugging

With Total/db, remote debugging is the only way to perform source level debugging optionally with GUI. The debug target is connected through a serial communication line such as the RS-232 to the debug host. There are no Ethernet or other types of communications available for remote kernel debugging (except SSPP). Like remote user process debugging, one can optionally use Insight for an intuitive GUI. SKDB works as the remote debug agent on the target.

Using sspp to Extend Serial Line Remote Debugging

In either remote user process or remote kernel/device driver debugging, if the remote debug target machine has only a serial port for communication with the debug host, this usually limits the physical distance between the two machines. LynuxWorks provides a solution to this: a third computer called a *proxy server*

running a server program `sspp` between the debug target and debug host will convert the serial line connection to TCP/IP communication so that the host machine can be located anywhere as long as there's a TCP/IP communication channel between the debug host and the proxy server.

Cross Debugging

In remote debugging, the debug host does not necessarily have to have the same CPU architecture and/or run the same operating system as the debug target. If the debug host has a different CPU architecture and/or runs a different operating system from the target, it is called *cross debugging*, whereas debugging with the same CPU architecture and operating system is called *native debugging*. Choose the cross debug host from the combinations of debug targets and hosts LynxOS supports.

Table 1-3: Remote Debugging

Component Target	Debug Host		Proxy Server	LynxOS Target
	Insight	GDB	sspp	Agent
User Process	Optional for GUI	Yes	Optional	GDBserver
Kernel/Device Driver	Optional for GUI	Yes	Optional	SKDB

Total/db Configuration Options

Total/db allows a wide range of flexibility in configuration, from stand-alone character based debugging to fully networked GUI based debugging. The following diagrams represent typical Total/db configurations.

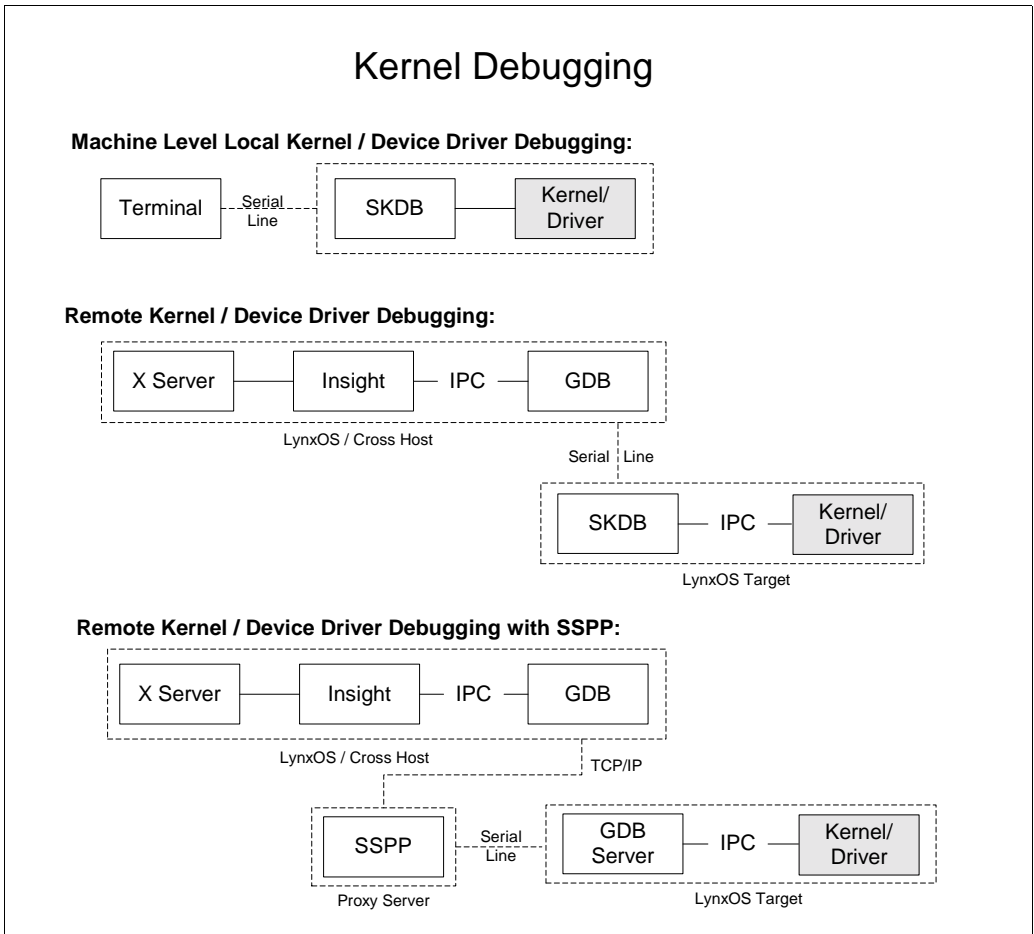


Figure 1-1: Kernel Debugging Configurations

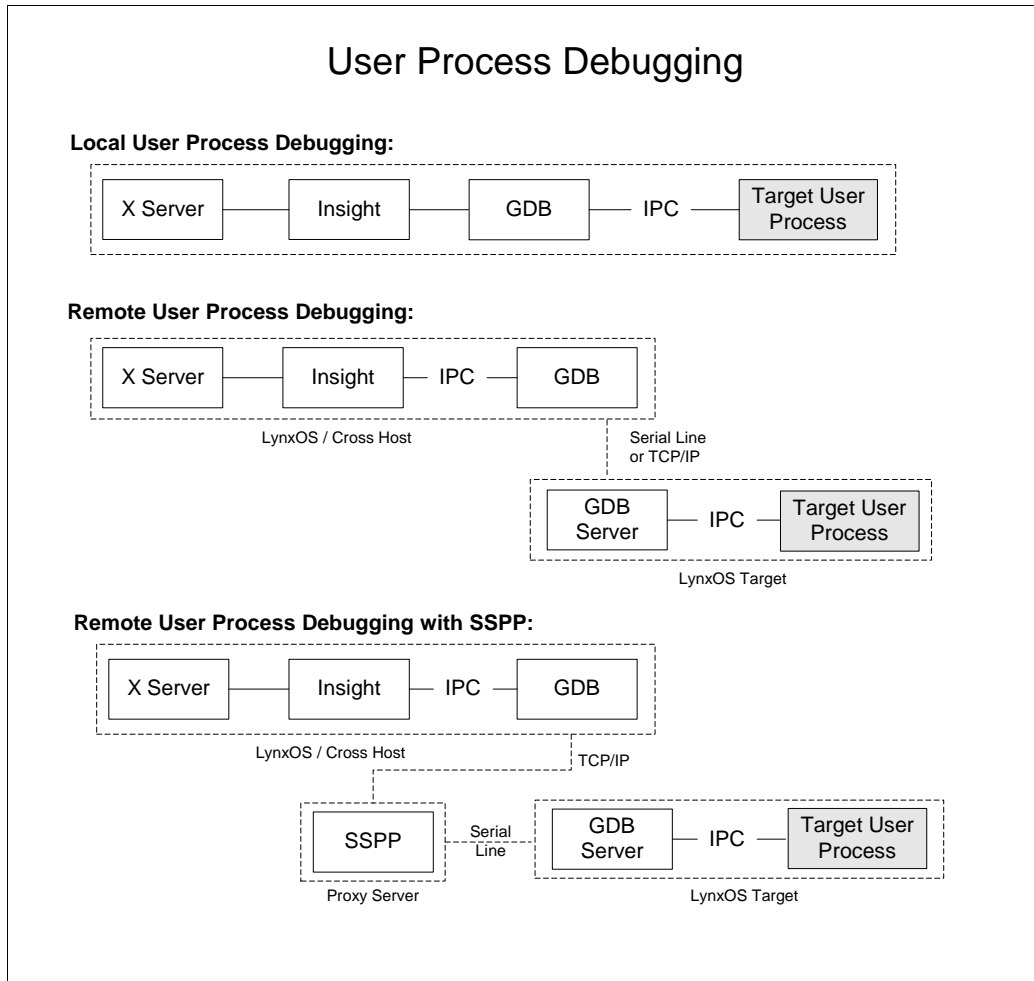


Figure 1-2: User Process Debugging Configurations

Supported Languages

Total/db currently supports the C and C++ programming languages plus the target's assembly language only.

Source Code

Parts of Total/db, namely GDB and Insight, are derived from public domain software. Though it is possible to obtain and build the source code for these programs, it may not work properly. Any such build is not supported by LynuxWorks.

CHAPTER 2 *Debugging with GDB*

This chapter is compiled from GNU's GDB manual: *Debugging with GDB*. Although GDB is flexible enough to support debugging of a variety of targets including different languages, LynuxWorks supports GDB only for debugging LynxOS target applications and drivers written in C, C++ or assembly languages in a LynxOS developed environment.

Additionally, see "LynxOS GDB Enhancements" on page 137 for extensions and enhancements made to the GDB.

The GNU Source Level Debugger

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

GDB can do four main 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.

You can use GDB to debug programs written in C and C++.

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.

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 76.

Prompt

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

NOTE: `set prompt` no longer adds a space for you after the prompt you set. This allows you to set a prompt which ends in a space or a prompt that does not.

```
set prompt newprompt
```

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

```
show prompt
```

Prints a line of the form: `Gdb's prompt is: your-prompt.`

Command Editing

GDB reads its input commands via the readline interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or vi-style inline editing of commands, `bash`-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 `set` command.

```
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 `GDBHISTFILE` environment variable, 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 exclamation point character (`!`). Because `!` 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 10 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 **Return** when you want to continue the output, or type **q** to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally, GDB knows the size of the screen from the 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 a period (`.`), and hexadecimal numbers begin with `0x`.

Numbers that begin with none of these are, by default, entered in base 10; likewise, the default format for displaying numbers is base 10. You can change the default base for both input and output with the `set radix` command.

```
set input-radix base
```

Sets the default base for numeric input. Supported choices for base are decimal 8, 10, or 16. `base` must itself be specified either unambiguously or using the current default radix; for example, any of `set radix 012`, `set radix 10`, or `set radix 0xa` set the base to decimal. On the other hand, `set radix 10` leaves the radix unchanged no matter what it was.

```
set output-radix base
```

Sets 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 which announce that the symbol table for a source file is being read; see `symbol-file` in “Commands to Specify Files” on page 106.

```
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 110).

```
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.

```
commandshow 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 and you had entered a `run` command you would see the following message on screen:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

If you are willing to unflinchingly face the consequences of your own commands, you can disable this “feature” 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.

Getting In and Out of GDB

The following material discusses invoking the debugger, choosing files, choosing modes, stopping the debugger and some essential shell commands.

The essentials are starting GDB and quitting GDB.

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

Invoking GDB

Invoke GDB by using the command, **gdb**. Once started, GDB reads commands from the terminal until you tell it 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 usual way to start GDB is with one argument, specifying an executable 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 and variables show.

```
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 and variables show.

```
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.

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

```
gdb -silent
```

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

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 you give are processed in sequential order. The order makes a difference when using the `-x` option.

Choosing Files

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

Many options have both long and short forms; both are shown in Table 2-1. GDB also recognizes the long forms if you truncate them, so long as enough of the

option is present to be unambiguous. (If you prefer, you can flag option arguments with `--` rather than `-`, though we illustrate the more usual convention.)

Table 2-1: Choosing Files

Long Entry Form	Short Entry Form	Command Definition
<code>-symbols file</code>	<code>-s file</code>	Read symbol table from file, <i>file</i> .
<code>-exec file</code>	<code>-e file</code>	Use file, <i>file</i> , as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.
<code>-se file</code>		Read symbol table from file, <i>file</i> , and use it as the executable file.
<code>-core file</code>	<code>-c file</code>	Use file, <i>file</i> , as a core dump to examine.
<code>-c number</code>		Connect to process ID number, as with the <code>attach</code> command (unless there is a file in <code>coredump</code> format named <i>number</i> , in which case <code>-c</code> specifies that file as a core dump to read).
<code>-command file</code>	<code>-x file</code>	Execute GDB commands from file <i>file</i> (see “Command Files” on page 118).
<code>-directory directory</code>	<code>-d directory</code>	Add <i>directory</i> to the path to search for source files.
<code>-r</code>	<code>-readnow</code>	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.

Choosing Modes

You can run GDB in various alternative modes—for example, in batch mode or quiet mode. Table 2-2 shows other available options.

Table 2-2: Choosing Modes

Long Entry Form	Short Entry Form	Command Definition
<code>-nx</code>	<code>-n</code>	Do not execute commands from any initialization files (normally called <code>.gdbinit</code>). Normally, the commands in these files are executed after all the command options and arguments have been processed (see “Command Files” on page 118).
<code>-quiet</code>	<code>-q</code>	Quiet. Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.
<code>-batch</code>		Run in batch mode. Exit with status 0 after processing all the command files specified with <code>-x</code> and all commands from initialization files, if not inhibited with <code>-n</code> . Exit with nonzero status if an error occurs in executing the GDB commands in the command files. Batch mode may be useful for running GDB as a filter. For example, to download and run a program on another computer, in order to make this more useful, the following message does not issue when running in batch mode (ordinarily, the message issues whenever a program running under GDB control terminates). Program exited normally.
<code>-cd directory</code>		Run GDB using <code>directory</code> as its working directory, instead of the current directory.

Table 2-2: Choosing Modes (Continued)

Long Entry Form	Short Entry Form	Command Definition
<code>-fullname</code>	<code>-f</code>	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 <code>\032</code> characters, followed by the file name, line number, and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two <code>\032</code> characters as a signal to display the source code for the frame.
<code>-b <i>bps</i></code>		Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.
<code>-tty <i>device</i></code>		Run using <i>device</i> for your program's standard input and output.

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 an Already-Running Process” on page 32).

Shell Commands

If you need to execute occasional `shell` commands during your debugging session, there is no need to leave or suspend GDB. Use the `shell` command to do this.

```
shell command string
```

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

```
make make-args
```

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

GDB Commands

The following material discusses the GDB commands.

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

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.

Straight brackets (`[]`) enclose optional parameters. Curly brackets (`{ }`) enclose choices or selections to be made. Neither of these brackets are typed in, but are inferred.

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 **Return** 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 and which it is unlikely you want to repeat.

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

GDB can also use **Return** in another way: to partition lengthy output, in a way similar to the common utility (see “Screen Size” on page 14). Because it is easy to use **Return** 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 118).

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** 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`, because that is the only `info` subcommand beginning with `bre`.

You can either use **Return** 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** 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, GDB sounds a bell. You can either supply more characters and try

again, or just use the **Tab** key a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with `make_`, but when you type `b make_` and use the **Tab** key, GDB just sounds the bell. Using the **Tab** key again displays all the function names in your program that begin with those characters. For example, you type `(gdb) make_b` and then use the **Tab** key. GDB sounds the bell; you use the **Tab** key again, to see the following display.

<code>make_a_section_from_file</code>	<code>make_envron</code>
<code>make_abs_section</code>	<code>make_function_type</code>
<code>make_blockvector</code>	<code>make_pointer_type</code>
<code>make_cleanup</code>	<code>make_reference_type</code>
<code>make_command</code>	<code>make_symbol_completion_list</code>
<code>(gdb) b make_</code>	

After displaying the available possibilities, GDB copies your partial input (in the example, `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, **M-?** rather than using **Tab** twice.

NOTE: **M-?** means using the **META** key (if there is one, or else, use **ESC**) and the **?** key. This is a command key sequence with which you may or may not be familiar.

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 **M-?** to request word completion, as in the following example:

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

Use the **M-?** 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:

```
(gdb) b bub
```

Use the **Tab** key at this point. GDB alters your input line to the following, and rings a bell.

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

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

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 example:

```
(gdb) help
List of classes of commands:
running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making 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.
```

```
help class
```

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

```
(gdb) help status Status inquiries.
List of commands:
show -- Generic command for showing things set with "set?"
info -- Generic command for printing status
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. Use *args* to specify the beginning of the command you want completed. For example: `complete i` results in the following.

```
info
inspect
ignore
```

This command is intentionally for use by GNU Emacs.

In addition to help, you can use the GDB `info` and `show` commands 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 were in the appropriate context. The listings under `info` and under `show` in the Index point to all the subcommands.

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 `$` 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 which of no 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.

Running Programs under GDB

The following material discusses running your programs with 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.

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 `-O` making it possible to debug optimized code.

We recommend that you always use `-g` whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

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: if you define a variable, but never use it, 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 `-g`, particularly on machines with instruction scheduling. If in doubt, recompile with `-g` alone, and if this fixes the problem, please report it to us as a bug (including a test case!).



CAUTION! The following discussions about your program’s arguments environment, working directory and input/output apply only if you start the debugged program locally from your GDB. If you attach GDB to an already running process, the parameters are already determined. If you start the application program remotely from a GDB subserver, the program arguments are given to GDB server’s command line, and the other parameters are inherited from the GDB server process

Starting Your Program

```
run  
r
```

Use the `run` command to start your program locally under GDB. You must first specify the program name with an argument to GDB or by using the file or `exec-file` command (see “Getting In and Out of GDB” on page 17 or “Commands to Specify Files” on page 106).

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.

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 30).

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 31).

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 32).



CAUTION! 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 36 for a 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 in “Examining Data” on page 67.

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 your user name, your home directory, your terminal type, and your search path for programs to run.

Usually you set up environment variables with the shell and they are inherited by all the other programs you run.

When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

`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 a colon (`:`) 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 `$cwd` string to refer to whatever is the current working directory at the time GDB searches the path. If you use a period (`.`) instead, it refers to the directory where you executed the `path` command. GDB replaces the period (`.`) 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`

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.

NOTE: 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 106).

`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 an Already-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 **Return** 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 106).

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 **Return** 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 16).

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.

The `kill` command is also useful if you wish to recompile and relink your program, because 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).

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 these facilities for debugging multithread programs:

- automatic notification of new threads
- `thread threadno`, a command to switch among threads
- `info threads`, a command to inquire about existing threads
- `thread apply [threadno][all] args`, a command to apply a command to a list of threads
- 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.

Whenever GDB detects a new thread in your program, it displays the target system's identification for the thread with a message in the `[New systag]`. *systag* form is a thread identifier whose form varies, depending on the particular system. For example, on LynxOS, you might see `[New process 35 thread 27]` when GDB notices a new thread. In contrast, on an SGI system, the *systag* is simply something like `process 368`, with no further qualifier.

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

```
info threads
```

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

1. The thread number assigned by GDB
2. The target system's thread identifier (*systag*)

3. The current stack frame summary for that thread

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

```
(gdb) info threads
3 process 35 thread 72 0x34e5 in sigpause ()
2 process 35 thread 23 0x34e5 in sigpause ()
*1 process 35 thread 13 main (argc=1, argv=0x7fffff8)
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 thread you selected, and its current stack frame summary:

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

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

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

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

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

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.

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 “Debugging an Already-Running Process” on page 32). From that point on you can debug the child process just like any other process to which you attached.

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 a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. 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
```

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

The following documentation provides more specific discussion on breakpoints, watchpoints, exceptions, and other information regarding stopping and continuing GDB.

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 37) 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.

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, 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 74).

GDB assigns a number to each breakpoint or watchpoint 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 ; 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 debugger convenience variable `$bpnum` records the number of the breakpoints you have set most recently; see “Convenience Variables” on page 83 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.

`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 at line *linenum* in the current source file. That file is the last file whose source text saw 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 56). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame.

This is similar to the effect of a `finish` command in the frame inside the selected frame—except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops. GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint.

This rule applies whether or not the breakpoint already existed when your program stopped.

`break...if cond`

Set a breakpoint with condition, *cond*; evaluate the expression, *cond*, each time the breakpoint is reached, and stop only if the value is non-zero—that is, if *cond*, evaluates as true. ‘...’ stands for one of the possible arguments described previously (or no argument) specifying where to break.

`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.

`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 used hardware breakpoints before setting new ones.

`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.



CAUTION! The current release of LynxOS does not support hardware assisted breakpoints, The above are provided only for information only.

`rbreak regex`

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

- Breakpoint Numbers
Type breakpoint or watchpoint.
- 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 72).

`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. GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of `long jmp` (in C programs). These internal breakpoints are assigned negative numbers, starting with `-1`; `info breakpoints` does not display them. You can see these breakpoints with the GDB maintenance command `maint info breakpoints`.

```
maint info breakpoints
```

Using the same format as `info breakpoints`, display both the breakpoints you have set explicitly, and those GDB is using for internal purposes.

Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

- `breakpoint`
Normal, explicitly set breakpoint
- `watchpoint`
Normal, explicitly set watchpoint

- `longjmp`
Internal breakpoint, used to handle correctly stepping through `longjmp` calls.
- `longjmp resume`
Internal breakpoint at the target of a `longjmp`.
- `until`
Temporary internal breakpoint used by the GDB `until` command.
- `finish`
Temporary internal breakpoint used by the GDB `finish` command.

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 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 `commands`, or two with `awatch` commands, but you cannot set one watchpoint with one command and the other with a different command. {No value for "GBDN" } 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
```

This command prints a list of watchpoints and breakpoints; it is the same as `info break`.



CAUTION! In multithread 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.

Hardware Watchpoints

Watchpoints can be implemented in Software or Hardware. Hardware watchpoints execute quicker than software watchpoints and allows the debugger to report a change in value at the exact instruction where the change occurred. Software watchpoints execute slower, and report a change in value in the statement following the change in value.

When setting a watchpoint, GDB attempts to set a hardware watchpoint first. If it is not possible to set a hardware watchpoint, a software watchpoint is set instead.

When issuing the `watch` command, and hardware watchpoints are set, GDB displays:

```
Hardware watchpoint num: expr
```

NOTE: Hardware Watchpoint support is not included in the default LynxOS kernel. To build the kernel for Hardware Watchpoint, Code Test, and Assertion support, use the following rule when running `make`:

```
# make all SYS_DEBUG=true
```

Breakpoints and Exceptions

Some languages, such as GNU C++, implement exception handling. You can use GDB to examine what caused your program to raise an exception, and to list the exceptions your program is prepared to handle at a given point in time.

`catch exceptions`

You can set breakpoints at active exception handlers by using the `catch` command. `exceptions` is a list of names of exceptions to catch.

You can use `info catch` to list active exception handlers (see “Information about a Frame” on page 60).

There are currently some limitations to exception handling in GDB:

If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits.

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, because 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 `__cp_push_exception` which has the following ANSI C interface:

```
/* addr is where the exception identifier is stored.
ID is the exception identifier. */
extern "C" void __cp_push_exception (void *value,
    void *type,
    void (*cleanup) (void *, int));
```

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__cp_push_exception` (see “Breakpoints, Watchpoints, and Exceptions” on page 36).

With a conditional breakpoint 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 spot your program when any of a number of exceptions are raised.

Deleting Breakpoints

It is often necessary to eliminate a breakpoint or watchpoint 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 or watchpoints 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 59). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

```
clear function  
clear filename: function
```

Delete any breakpoints set at entry to the function, *function*.

```
clear linenum  
clear filename: linenum
```

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

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

Delete the breakpoints or watchpoints 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 or watchpoint you might prefer to disable it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can enable it again later. You disable and enable breakpoints and watchpoints 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 or watchpoints if you do not know which numbers to use. A breakpoint or watchpoint can have any of four different states of enablement:

- Enabled
The breakpoint stops your program. A breakpoint set with the `break` command starts out in this state.
- Disabled
The breakpoint has no effect on your program.
- Enabled once
The breakpoint stops your program, but then becomes disabled. 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 and watchpoints.

```
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 37), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the previously discussed commands. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints (see “Continuing and Stepping” on page 50).

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 68). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is true.

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

Conditions are also accepted for watchpoints; you may not need them, because 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 47).

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 37). 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 or watchpoint number, *bnum*. After you set a condition, breakpoint *bnum* stops your program only if the value of *expression* is true (non-zero, 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 68).

```
condition bnum
```

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

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

```
ignore bnum count
```

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

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

When you use *continue* to resume execution of your program from a breakpoint, you can specify an *ignore count* directly as an argument to *continue*, rather than using *ignore* (see “Continuing and Stepping” on page 50).

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

You could achieve the effect of the ignore count with a condition such as `$foo-- <= 0` using a debugger convenience variable that is decremented each time (see “Convenience Variables” on page 83).

Breakpoint Command Lists

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

```
commands [bnum]  
...command-list...  
end
```

Specify a list of commands for breakpoint number, *bnum*. The commands themselves appear on the following lines.

Type a line containing just `end` to terminate the commands. To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands.

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

Using `Return` as a means of repeating the last GDB command is disabled within a `command-list`.

You can use breakpoint commands to start your program up again.

Simply use the `continue` command, or `step`, or any other command that resumes execution.

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

If the first command you specify in a command list is `silent`, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue.

If none of the remaining commands print anything, you see no sign that the breakpoint was reached.

`silent` is meaningful only at the beginning of a breakpoint command list.

The commands `echo`, `output`, and `printf` allow you to print precisely controlled output, and are often useful in silent breakpoints (see “Commands for Controlled Output” on page 119.)

For example, the following 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 ",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 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
set x=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 . The first two options are always [0] cancel and [1] all. Typing **1** sets a breakpoint at each definition of *function*, and typing **0** aborts the break command without setting any new breakpoints.

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

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] ;cc. String:file line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
>2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line .578
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the 'delete' command to delete unwanted breakpoints.
(gdb)
```

Continuing and Stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more “step” of your program, where “step” may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If due to a signal, you may want to use `handle`, or use `signal 0` to resume execution; see “Signals” on page 53.)

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

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument, *ignore-count*, allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore`.

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

The synonyms, `c` and `fg` are provided purely for convenience, and have exactly the same behavior as `continue`.

To resume execution at a different place, you can use `Return` (see “Returning from a Function” on page 105) to go back to the calling function; or `jump`. See “Continuing at a Different Address” on page 103. to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint 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`.



CAUTION! 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.

The `step` command only stops at the first instruction of a source line. This prevents 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 saw 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 stacks 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 105).

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. 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 37).

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 74.

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 synchronous 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 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 are:

`nostop`

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

`stop`

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

`print`

GDB should print a message when this signal happens.

`noprint`

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

`pass`

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

`nopass`

GDB should not allow your program to see this signal.

When a signal stops your program, the signal is not visible 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 was the signal. To prevent this, you can continue with `signal 0`.

Stopping and Starting Multithread Programs

When your program has multiple threads (see “Debugging Programs with Multiple Threads” on page 34), 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, as the following example shows.

```
(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 such as `step` or `next`.

In particular, GDB cannot single-step all threads in lockstep. Because 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.

Examining the Stack

The following documentation discusses GDB, stack frames and other related topics.

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

Each time your program performs a function call, information about the call is generated. 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.

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 whichever frame you are interested in (see “Selecting a Frame” on page 59).

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command (see “Information about a Frame” on page 60).

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 `-fomit-frame-pointer gcc` option 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, normally **Ctrl-c**.

```
backtrace n
bt n
```

Similar, but print only the innermost *n* frames.

```
backtrace -n
bt -n
```

Similar, but print only the outermost *n* frames.

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

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use `set print address off`. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number. Here is an example of a backtrace. It was made with the command `bt 3`, so it shows the innermost three frames.

```
#0      m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
        at builtin.c:993
#1      0x6e38 in expand_macro (sym=0x2b600) at
        macro.c:242
#2      0x6840 in expand_token (obs=0x0, t=177664,
        td=fffb08)
        at macro.c:71
(More stack frames to follow...)
```

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

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. Here are the commands 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, it needs three addresses: a register stack pointer, a program counter, and a memory stack pointer.

```
up n
```

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

```
down n
```

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

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the

source file and line number of execution in that frame. The second line shows the text of that source line. For instance, use the following as an example.

```
(
g
d
b
)
u
p

#      0x22f0 in main (argc=1, argv=0xf7ffbf4,
1      env=0xf7ffbf4) at env.c:10

1      read_input_file (argv[i]);
0
```

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 62).

```
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, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated `f`. With an argument, this command is used to select a stack frame (see “Selecting a Frame” on page 59).

```
info frame
info f
```

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

- the address of the frame

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

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

`info frame addr`

`info f addr`

Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the `frame` command (see “Selecting a Frame” on page 59).

`info args`

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

`info locals`

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

`info catch`

Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the `up`, `down`, or `frame` commands); then type: **info catch**. See “Breakpoints, Watchpoints, and Exceptions” on page 36.

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:

```
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.

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

Examining Source Files

GDB can print parts of your program's source, because the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see "Selecting a Frame" on page 59), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

See the following documentation for more specific discussion on source files and GDB.

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 120).

Printing Source Lines

To print lines from a source file, use the `list` command (abbreviated `l`). By default, 10 lines are printed. There are several ways to specify what part of the file you want to print. The following are the forms of the `list` command 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 56), this prints lines centered around that line.

```
list -
```

Print lines just before the lines last printed.

By default, GDB prints 10 source lines with any of these forms of the `list` command. You can change this using `set listsize`.

```
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.

Repeating a `list` command using **Return** 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 `-`; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the `list` command expects you to supply zero, one or two *linespecs* which specify source lines. There are several ways of writing *linespecs*, but the effect is always to specify some source line. Here is a complete description of 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 are *linespecs*.

```
list, last
```

Print lines ending with *last*.

`list first,`

Print lines starting with *first*.

`list +`

Print lines just after the lines last printed.

`list -`

Print lines just before the lines last printed.

`list`

As described for `list` in the previous descriptions.

The following are the ways of specifying a single source line—all the kinds of linespec.

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*. *address* may be any expression.

Searching Source Files

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

```
forward-search regex
```

```
search regex
```

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

```
reverse-search regex
```

The `reverse-search regex` command checks each line, starting with the one before the last line listed and going backward, for a match for *regex*. 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. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB 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.

NOTE: 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 a colon (:) 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` is not the same as a period (.)—the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

```
directory
```

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

```
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 by the following methods.

- 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 `info line` command to map source lines to program addresses (and vice versa), and the `disassemble` command 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 62).

For instance, we can use `info line` to discover the location of the object code for the first line of function, `m4_changequote`, as in the following example.

```
(gdb) info line m4_changecom
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can 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 72). Also, this address is saved as the value of the convenience variable, `$_` (see “Convenience Variables” on page 83).

`disassemble`

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

We can use `disassemble` to inspect the object code range shown in the last `info line` example (the example shows SPARC machine instructions):

```
(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>: 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.
```

Examining Data

The following material relates to examining data using GDB.

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym, `inspect`. It evaluates and prints the value of an expression of the language your program is written in. See “Using GDB with Different Languages” on page 86.

```
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 71).

```
print /f
```

If you omit *exp*, GDB displays the last value again (from the *value history*; see “Value History” on page 82). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See “Examining Memory” on page 72.

If you are interested in information about types, or about how the fields of a struct or class are declared, use the `ptype exp` command rather than `print`. See “Examining the Symbol Table” on page 99.

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

NOTE: Because C is so widespread, most of the expressions shown in examples in this manual are in C. See “Using GDB with Different Languages” on page 86.

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

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

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

::

:: allows you to specify a variable in terms of the file or function where it is defined. See “Program Variables” on page 69.

@

@ is a binary operator for treating parts of memory as arrays. See “Artificial Arrays” on page 70.

{ 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 59); they must be either global (or static) or 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 means that you can examine and use the variable, *a*, whenever your program is executing within the function, *foo*, but you can only use or examine the variable, *b*, while your program is executing inside the block where *b* is declared.

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
```

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

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



CAUTION! Occasionally, a local variable may appear to have the wrong value at certain points in a function—just after entry to a new scope, and just before exit. You may see this problem when you are stepping by machine instructions. This is because, on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

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 82), 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 a size to fill the value, as `sizeof(value)/sizeof(type)` as the following example shows.

```
(gdb) p/x (short[])0x1234567
$2 = {0x1234, 0x5678}
```

Sometimes, the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see “Convenience Variables” on page 83) as a counter in an expression that prints the first interesting value, and then repeat that expression using **Return**. 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 type:

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

(At this point, use **Return** twice.)

Output Formats

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

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

Letter Value	Definition
x	Regard the bits of the value as an integer, and print the integer in hexadecimal.
d	Print as integer in signed decimal.
u	Print as integer in unsigned decimal.
o	Print as integer in octal.
t	Print as integer in binary. The letter 't' stands for "two" ('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 72).

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

c	Regard as an integer and print it as a character constant.
f	Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see "Registers" on page 84), 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.

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/ nfuaddr
x addr
```

`x` Use the `x` command to examine memory.

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

n, the repeat count

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

f, the display format

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

u, the unit size

The unit size is shown in the following table.

Table 2-3: Unit Size

Type	Unit
b	Bytes
h	Half words (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 68. 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/3uh0x54320` is a request to display three half words (`h`) of memory, formatted as unsigned decimal integers (`u`), starting at address

`0x54320.x/4xw$sp` prints the four words (*w*) of memory above the stack pointer (here, `$sp`; (see “Registers” on page 84) in hexadecimal (*x*).

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

Even though the unit size *u* is ignored for the formats *s* and *i*, you might still want to use a count *n*. For example, `3i` specifies that you want to see three machine instructions, including any operands. The `disassemble` command gives an alternative way of inspecting machine instructions; see “Source and Machine Code” on page 66.

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/3iaddr`, you can inspect the next seven with just `x/7`. If you use **Return** 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 68).

`display` does not repeat if you press **Return** 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 71).

```
display/fmt addr
```

For `fmt` `i` or `s`, or including a unit-size or a number of units, add the expression, `addr`, as a memory address to be examined each time your program stops. Examining means in effect doing `x/fmt addr` (see “Examining Memory” on page 72).

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 84).

```
undisplay dnums...
```

```
delete display dnums...
```

Remove item numbers `dnums` from the list of expressions to `display`.

`undisplay` does not repeat if you use **Return** 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
#0set_quotes (lq=0x34c78 "<<", rq=0x34c88 "")
at input.c:530 530
530if (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
#0set_quotes (lq="<<", rq="")at input.c:530
530if (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
```

Displays 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
```

Tell GDB to only 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`. This interprets the address in symbolic form. For instance, the following 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
```



CAUTION! 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:

```
$l={
  next = 0x0
  flags = {
    sweet = 1,
    sour = 1
  },
  530meat = 0x54 "Pork"
```

```
set print pretty off
```

Cause GDB to print structures in a compact format, like the following example:

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

This is the default format.

```
show print pretty
```

Show which format GDB is using to print structures.

```
set print sevenbit-strings on
```

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

```
set print sevenbit-strings off
```

Print full eight-bit characters. This allows the use of more international character sets, and is the default.

```
show print sevenbit-strings
```

Show whether or not GDB is printing only seven-bit characters.

```
set print union on
```

Tell GDB to print unions which are contained in structures. This is the default setting.

```
set print union off
```

Tell GDB not to print unions which are contained in structures.

```
show print union
```

Ask GDB whether or not it will print unions which are contained in structures. For instance, consider the following example's declarations.

```
typedef enum {Tree, Bug} Species; typedef enum {Big_tree, Acorn,
Seedling} Tree_forms; typedef enum {Caterpillar, Cocoon, Butterfly}
Bug_forms;
struct thing {
Species it;      union {      Tree_forms tree;      Bug_forms bug;
} form;
};

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

The example has `set print union on` in effect having
`p foo` printing the following result.

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

With `set print union off` in effect, it would print the following result.

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

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. The default is on.

```
show print demangle
```

Show whether C++ names are printed in mangled or demangled form.

```
set print asm-demangle
set print asm-demangle on
```

Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

```
show print asm-demangle
```

Show whether C++ names in assembly listings are printed in mangled or demangled form.

```
set demangle-style style
```

Choose among several encoding schemes used by different compilers to represent C++ names. 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.

lucid

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

arm

Decode using the algorithm in the *C++ Annotated Reference Manual*.

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

foo

Show the list of 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 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*. This allows you to refer to them in other expressions. Values are kept until the symbol table is reread or discarded (for example with the `file` or `symbol-file` commands). When the symbol table changes, the value history is discarded, because because 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; `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—which you can do by just using **Return**.

Note that 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 **Return** to repeat `show values n` has exactly the same effect as **SHOW VALUES +**.

Convenience Variables

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

Convenience variables are prefixed with `$`. Any name preceded by `$` can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see “Registers” on page 84). Value history references, in contrast, are *numbers* preceded by `$` (see “Value History” on page 82).

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 as `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 **Return**.

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

`$_`

The `$_` variable is automatically set by the `x` command to the last address examined (see “Examining Memory” on page 72). Other commands which provide a default address for `x` to examine also set `$_` 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 `$_exitcode` variable 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*. As discussed in the following, 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.
```

This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use **Return** (see “Returning from a Function” on page 105).

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 59). 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.

```
set rstack_high_address address
```

On AMD 29000 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”. This may result in GDB referencing memory locations that do not 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 the ARM and x86 machines.

Using GDB with Different Languages

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

Language-specific information is built into GDB for some languages, allowing you to express operations like the 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*.

See the following documentation for more specific discussion on languages that GDB accommodates.

NOTE: Although GDB is designed to support multiple languages, LynxWorks currently supports only GDB for C, C++, and assembly languages.

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 on.

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.

C source file

`.c`

C++ source file

```
.C  
.cc  
.cxx  
.cpp  
.cp  
.c++
```

Assembler source file *

```
.s  
.S
```

*Assembler source files behave almost like C, but GDB does not skip over function prologues when stepping.

Setting the Working Language

If you allow GDB to set the language automatically, expressions are interpreted the same way in your debugging session and your program. If you wish, you may set the language manually. To do this, issue the `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, type `set language`.

Setting the language manually prevents GDB from updating the working language automatically. This can lead to confusion if you try to debug a program when the working language is not the same as the source language, when an expression is acceptable to both languages—but means different things. For instance, if the current source file were written in C, and GDB was parsing Modula-2, a command such as `print a =b +c` might not have the effect you intended. In C, this means to add `b` and `c` and place the result in `a`. The result printed would be the value of `a`. In Modula-2, this means to compare `a` to the result of `b+c`, yielding a *Boolean* value.

Having GDB Infer the Source Language

To have GDB set the working language automatically, use `set language local` or `set language auto`. GDB then infers the working language. That is, when your program stops in a frame (usually by encountering a breakpoint), GDB sets the working language to the language recorded for the function in that frame. If the language for a frame is unknown (that is, if the function or block corresponding to the frame was defined in a source file that does not have a recognized extension), the current working language is not changed, and GDB issues a warning.

This may not seem necessary for most programs, which are written entirely in one source language. However, program modules and libraries written in one source language can be used by a main program written in a different source language. Using `set language auto` in this case frees you from having to set the working language manually.

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 60 to identify the other information listed here.

`info source`

Display the source language of this source file. See “Examining the Symbol Table” on page 99.

Type and Range Checking



CAUTION! In this release, the GDB commands for type and range checking are included, but they do not yet have any effect. This section documents the intended facilities.

Some languages are designed to guard against you making seemingly common errors through a series of compile and run-time checks. These include checking the type of arguments to functions and operators, and making sure mathematical overflows are caught at run time. Checks such as these help to ensure 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.

GDB can check for conditions like the previous if you wish. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation via the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your program's source language. See "Supported Languages," later in this chapter for the default settings of supported languages.

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 two following examples.

```
1 +2 3
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" 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" for the default settings for each language.

```
set check type on
set check type off
```

Set type checking on or off, overriding the default setting for the current working language. Issue a warning if the setting does not match the language default. If any type mismatches occur in evaluating an expression while type checking is on, GDB prints a message and aborts evaluation of the expression.

```
set check type warn
```

Cause the type checker to issue warnings, but to always attempt to evaluate the expression. Evaluating the expression may still be impossible for other reasons. For example, GDB cannot add numbers and structures.

```
show type
```

Show the current setting of the type checker, and whether or not GDB is setting it automatically.

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

$$m + 1 \leq s$$

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. See “Supported Languages” for further details on specific languages. GDB provides some 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” 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 4 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 68) can be used with the constructs of any supported language. The following sections detail 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; please look to these for a language reference or tutorial.

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 and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with the GNU C++ compiler, 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 for Debugging Your Program or GNU CC” 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, listed in order of increasing precedence:

<code>,</code>	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.
<code>=</code>	Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.
<code>op=</code>	Used in an expression of the form <code>a op=b</code> , and translated to <code>a= a opb</code> . <code>op=</code> and <code>=</code> have the same precedence. <code>op</code> is any one of the operators <code> </code> , <code>^</code> , <code>&</code> , <code><<</code> , <code>>></code> , <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> .
<code>? :</code>	The ternary operator. <code>a?b: c</code> can be thought of as: if <code>a</code> , then <code>b</code> , else, <code>c</code> . <code>a</code> should be of an integral type.
<code> </code>	Logical OR. Defined on integral types.
<code>&&</code>	Logical AND. Defined on integral types.
<code> </code>	Bitwise OR. Defined on integral types.
<code>^</code>	Bitwise exclusive-OR. Defined on integral types.
<code>&</code>	Bitwise AND . Defined on integral types.

<code>==, !=</code>	Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.
<code><, >, <=, >=</code>	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.
<code><<, >></code>	Left shift, and right shift. Defined on integral types.
<code>@</code>	The GDB “artificial array” operator (see “Expressions” earlier in this chapter).
<code>+, -</code>	Addition and subtraction. Defined on integral types, floating-point types and pointer types.
<code>*, /, %</code>	Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.
<code>++, --</code>	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.
<code>*</code>	Pointer dereferencing. Defined on pointer types. Same precedence as <code>++</code> .
<code>&</code>	Address operator. Defined on variables. Same precedence as <code>++</code> .

For debugging C++, GDB implements a use of `&` beyond what is allowed in the C++ language itself: you can use `&(&ref)` (or, if you prefer, `&&ref`) to examine the address where a C++ reference variable (declared with `&ref`) is stored.

<code>-</code>	Negative. Defined on integral and floating-point types. Same precedence as <code>++</code> .
<code>!</code>	Logical negation. Defined on integral types. Same precedence as <code>++</code> .
<code>~</code>	Bitwise complement operator. Defined on integral types. Same precedence as <code>++</code> .

<code>.</code> , <code>-></code>	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.
<code>[]</code>	Array indexing. <code>a[i]</code> is defined as <code>*(a+i)</code> . Same precedence as <code>-></code> .
<code>()</code>	Function parameter list. Same precedence as <code>-></code> .
<code>::</code>	C++ scope resolution operator. Defined on <code>struct</code> , <code>union</code> , and class types.
<code>:::</code>	Doubled colons also represent the GDB scope operator (“Expressions” on page 68). Same precedence as <code>::</code> .

C and C++ Constants

GDB allows you to express the constants of C and C++ in the following ways:

Integer constants are a sequence of digits. Octal constants are specified by a leading `0` (i.e., zero), and hexadecimal constants by a leading `0x` or `0X`. Constants may also end with a letter, `l`, specifying that the constant should be treated as a *long* value.

Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: `e[[+] | -]nnn`, where `nnn` is 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; or of the form `\x`, where `x` is a predefined special character—for example, `\n` for newline.

String constants are a sequence of character constants surrounded by double quotes (`" "`).

Pointer constants are an integral value. You can also write pointers to constants using the C operator, `&`.

Array constants are comma-separated lists surrounded by braces `{` and `}`; for example, `{1, 2, 3}` is a three-element array of integers, `{{1, 2}, {3, 4}, {5, 6}}` is a three-by-two array, and `{&"hi", &"there", &"fred"}` is a three-element array of pointers.

C++ Expressions

GDB expression handling has a number of extensions to interpret a significant subset of C++ expressions.



CAUTION! GDB can only debug C++ code if you compile with the GNU C++ compiler. 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 GNU CC, you can use the `'-gstabs'` option to request `stabs` debugging extensions explicitly.) Where the object code format is standard COFF or DWARF in ELF, on the other hand, most of the C++ support in GDB does not work.

Member function calls are allowed; you can use expressions like

```
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 specified `set print address off`.

GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another,

you can use `::` repeatedly if necessary, for example in an expression such as `scope1::scope2::name`. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see “Program Variables” earlier in this chapter).

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,” earlier in this chapter, for further 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, it appears as `{...}`.

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function (see “Expressions” on page 68).

GDB features for C++

Some GDB commands are particularly useful with C++, and some are designed specifically for use with C++. The following is a summary:

`breakpoint menus`

When you want a breakpoint in a function whose name is overloaded, GDB `breakpoint menus` help you specify which function definition you want (see “Breakpoint Menus,” earlier in this chapter).

`rbreakregex`

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 37.

`catchexceptions`

`info catch`

Debug C++ exception handling using these commands. See “Breakpoints and Exceptions,” earlier in this chapter.

`ptypetypername`

Print inheritance relationships as well as other information for type `typename`. See “Examining the Symbol Table” on page 99.

`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,” earlier in this chapter.

`set print object`

`show print object`

Choose whether to print derived (actual) or declared types of objects. See “Print Settings,” earlier in this chapter.

`set print vtbl`

`show print vtbl`

Control the format for printing virtual function tables. See “Print Settings,” earlier in this chapter.

Overloaded Symbol Names

You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type `symbol(types)` rather than just `symbol`. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See “Command Completion,” earlier in this chapter, for details on how to perform this function.

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,” earlier in this chapter), or by one of the file-management commands (see “Commands to Specify Files,” later in this chapter).

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,” earlier in this chapter). File names are recorded in object files as debugging symbols, but GDB would ordinarily parse a typical file name, such as `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 stack-frame offset at which the variable is always stored.

NOTE: The contrast with `print &symbol` does not work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

```
whatis 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 68).

`what is`

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`.

`ptype exp`

ptype

Print a description of the type of expression *exp*. *ptype* differs from `what is` 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 give the following output.

```
(gdb) what is v
type = struct complex
(gdb) ptype v
type = struct complex {
double real;
double imag;
```

As with `what is`, 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 `what is`, it does not print a detailed description; second, it lists all source files where a type is defined.

`info source`

Show the name of the current source file—that is, the source file for the function containing the current point of execution—and the language it was written in.

```
info sources
```

Print the names of all source files in your program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols will be read when needed.

```
info functions
```

Print the names and data types of all defined functions.

```
info functions regexp
```

Print the names and data types of all defined functions whose names contain a match for regular expression, *regexp*. Thus, `info fun step` finds all functions whose names include `step`; `info fun ^step` finds those whose names start with `step`.

```
info variables
```

Print the names and data types of all variables that are declared outside of functions (i.e., excluding local variables).

```
info variables regexp
```

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

Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. 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 symbolsfilename
```

```
maint print psymbolsfilename
```

```
maint print msymbolsfilename
```

Write a dump of debugging symbol data into the file, *filename*. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included.

If you use `maint print symbols`, GDB includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols GDB has read.

You can use the `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,” later in this chapter 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.

See the following documentation for more details.

Assignment to Variables

To alter the value of a variable, evaluate an assignment expression (see “Expressions” on page 68). 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 86.

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 82). 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 command `set width`:

```
(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 68). 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 line, *linespec*. Execution stops again immediately if there is a breakpoint there. See “Printing Source Lines” on page 62 for a description of the different forms of *linespec*.

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 50).

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 Your Program a Signal

```
signal signal
```

Resume execution where your program stopped, but immediately give it the signal *signal*. *signal* can be the name or the number of a signal. For example, on many systems `signal 2` and `signal SIGINT` are both ways of sending an interrupt signal.

Alternatively, if *signal* is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would 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** a second time after executing the command.

Invoking the `signal` command is not the same as invoking the `kill` utility from the shell. Sending a signal with `kill` causes GDB to decide what to do with the signal, depending on the signal handling tables (see “Signals” on page 53). The `signal` command passes the signal directly to your program.

Returning from a Function

```
return  
return expression
```

You can cancel execution of a function call with the `return` command. If you give an *expression* argument, its value is used as the function’s return value.

When you use `return`, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to `return`.

This pops the selected stack frame (see “Selecting a Frame” on page 59), 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 50) resumes execution until the selected stack frame returns naturally.

Calling Program Functions

```
call expr
```

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

You can use this variant of the `print` command if you want to execute a function from your program, but without cluttering the output with `void` returned values. If the result is not `void`, it is printed and saved in the value history.

A new 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) 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, you can specify that explicitly with the `set write` command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

```
set write on
set write off
```

If you specify `set write on`, GDB opens executable and core files for both reading and writing; if you specify `set write off` (the default), GDB opens them 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 provides more details on command specification and symbol files with GDB.

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 "Getting In and Out of GDB" on page 17).

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.

`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** 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 16.)

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]
file filename[-readnow]
```

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.

You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program. The auxiliary symbol file for a program called `myprog` is called `myprog.syms`. Once this file exists (so long as it is newer than the corresponding executable), GDB always attempts to use it when you debug `myprog`; no special options or commands are needed.

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

```
core-file [filename]
```

Specify the whereabouts of a core dump file to be used as the “contents of memory”. Traditionally, core files contain only some parts of the address space of the process that generated them; GDB can access the executable file itself for other parts.

`core-file` with no argument specifies that no core file is to be used.

NOTE: 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 33).

```
loadfilename
```


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, e.g., `load` also records the *filename* symbol table in GDB, like the `add-symbol-file` command.

If your GDB does not have a `load` command, attempting to execute it gets the error message “You can’t do that when your target is...”

The file is loaded at whatever address is specified in the executable. For some object file formats, you can specify the load address when you link the program; for other formats, like *a.out*, the object file format specifies a fixed address.

`load` does not repeat if you use **Return** 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**.

You can use the `-readnow` option, just as with the `symbol-file` command, to change how GDB manages the symbol table information for *filename*.

`section`

The `section` command changes the base address of section, `SECTION`, of the exec file to `ADDR`. This can be used if the exec file does not contain section addresses (such as in the *a.out* format), or when the addresses specified in the file itself are wrong. Each section must be changed separately. The `info files` command 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 112), 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.

```
info share
info sharedlibrary
```

Print the names of the shared libraries which are currently loaded.

```
sharedlibraryregex
shareregex
```

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 bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers.

If you are interested in seeing information about ill-constructed symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the `set complaints` command as shown in “Optional Warnings and Messages” on page 16.

The messages currently printed, and their meanings, include the following.

```
inner block not inside outer block in symbol
```

The symbol information shows where symbol scopes begin and end (such as at the start of a function or a block of statements). This error indicates that an inner scope block is not fully contained in its outer scope blocks.

GDB circumvents the problem by treating the inner block as if it had the same scope as the outer block. In the error message, *symbol* may be shown as “(don't know)” if the outer block is not a function.

block at *address* out of order

The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so.

GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine what source file is affected by specifying `set verbose on`. See “Optional Warnings and Messages” on page 16.

bad block start address patched

The symbol information for a symbol scope block has a start address smaller than the address of the preceding source line. This is known to occur in the SunOS 4.1.1 (and earlier) C compiler.

GDB circumvents the problem by treating the symbol scope block as starting on the previous source line.

bad string table offset in symbol *n*

Symbol number *n* contains a pointer into the string table which is larger than the size of the string table. GDB circumvents the problem by considering the symbol to have the name, `foo`, which may cause other problems if many symbols end up with this name.

unknown symbol type `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_symtab` and examine `*bufp` to see the symbol.

stub type has NULL name

GDB could not find the full definition for a struct or class.

const/volatile indicator missing
ok if using `g++ vl.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.

The following material provides more details on GDB specification.

Active Targets

There are three classes of targets: processes, core files, and executable files.

GDB can work concurrently on up to three active targets, one in each class. This allows you to (for example) start a process and inspect its activity without abandoning your work on a core file.

For example, if you execute `gdb a.out`, then the executable file, `a.out`, is the only active target. If you designate a core file as well—presumably from a prior run that crashed and coredumped—then GDB has two active targets and uses them in tandem, looking first in the corefile target, then in the executable file, to satisfy requests for memory addresses. (Typically, these two classes of target are complementary, since core files contain only 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 106). To specify as a

target a process that is already running, use the `attach` command (see “Debugging an Already-Running Process” on page 32).

Commands for Managing Targets

`target type parameters`

Connects the GDB host environment to a target machine or process. A target is typically a protocol for talking to debugging facilities. You use the argument, *type*, to specify the type or protocol of the target machine.

Further *parameters* are interpreted by the target protocol, but typically include things like device names or host names to connect with, process numbers, and baud rates.

The `target` command does not repeat if you 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 106).

`help target name`

Describe a particular target, including any parameters necessary to select it.

`set gnutarget args`

GDB uses its own library, BFD, to read your files. GDB knows whether it is reading an *executable*, a *core*, or a *.o file*; however you can specify the file format with the `set gnutarget` command.

Unlike most `target` commands, with `gnutarget`, the `target` refers to a program, not a machine.



CAUTION! To specify a file format with `set gnutarget`, you must know the actual BFD name. See “Commands to Specify Files” on page 106.

`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 this message:

```
The current BDF target is "auto".
```

The following are some common targets (available, or not, depending on the GDB configuration).

```
target exec program
```

An executable file, target `exec Substitute Text`, is the same as `exec-file program`.

```
target core filename
```

A core dump file, target `core filename`, is the same as `core-file filename`.

```
target remote dev
```

Remote serial target in GDB-specific protocol. The argument, `dev`, specifies what serial device to use for the connection (e.g., `/dev/ttya`); see “Remote Debugging” on page 114. `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.

Different targets are available on different configurations of GDB; your configuration may have more or fewer targets.

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.

Using the gdbserver program

`gdbserver` is a control program for UNIX-like systems, which allows you to connect your program with a remote GDB via `target remote`—but without linking in the usual debugging stub.

GDB and `gdbserver` communicate via either a serial line or a TCP connection, using the standard GDB remote serial protocol.

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 via TCP. The `host:2345` argument means that `gdbserver` is to expect a TCP connection from machine `host` to local TCP port 2345. (Currently, 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. If you choose a port number that conflicts with another service, `gdbserver` prints an error message and exits.

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 TCP port descriptor in the form, `host:port`. For example, `(gdb) target remote /dev/ttyb` communicates with the server via serial line, `/dev/ttyb`.

`(gdb) target remote target:2345` communicates via a TCP connection to port 2345 on host, `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.”

Stored Command Sequences

Aside from breakpoint commands (see “Breakpoint Command Lists” on page 47), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

User-Defined Commands

A user-defined command is a sequence of GDB commands to which you assign a new name as a command. This is done with the `define` command. User commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command via

`$arg0 ... $arg9`. A trivial example is the following:

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

To execute the command use the following:

```
adder 1 2 3
```

This defines the `adder` command, which prints the sum of its three arguments.

NOTE: The arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior function 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 (non-zero). 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 *commandname* command so that it can be accessed by `help`. The *commandname* command must already be defined. This command reads lines of documentation just as `define` reads the lines of the command definition, ending with `end`. After the `document` command is finished, `help` on command, *commandname*, displays the documentation you have written. You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

`help user-defined`

List all user-defined commands, with the first line of the documentation (if any) for each.

`show user`

`show user commandname`

Display the GDB commands used to define *commandname* (but not its documentation). If no *commandname* is given, display the definitions for all user-defined commands.

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command. If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

User-Defined Command Hooks

You may define hooks, which are a special kind of user-defined command. Whenever you run the `foo` command, if the user-defined `hook-foo` command exists, it is executed (with no arguments) before that command. In addition, a pseudo-command, `stop`, exists. Defining `hook-stop` makes the associated commands execute every time execution stops in your program: before breakpoint commands are run, displays are printed, or the stack frame is printed. For example, to ignore `SIGALRM` signals while single-stepping, 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, e.g., `backtrace` rather than `bt`. If an error occurs during the execution of your hook, execution of GDB commands stops and GDB issues a prompt (before the command that you actually 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*. These are 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 20. 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 commands useful for generating exactly the output you want.

```
echo text
```

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

NOTE: No newline is printed unless you specify one.

In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments. To print `and foo =`, use the `echo \ and foo = \` command. A backslash at the end of text can be used, as in C, to continue the command on to subsequent lines.

Consider the following example.

```
echo This is some text \  
which is continued \  
onto several lines.
```

The previous example shows input that produces the same output as the following.

```
echo This is some text  
echo which is continued  
echo onto several lines.
```

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 68 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 71.

`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 example shows.

```
printf "foo, bar-foo = 0x%x, 0x%x ", 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.

Using GDB under GNU Emacs

A special interface allows you to use GNU Emacs to view (and edit) the source files for the program you are debugging with GDB.

To use this interface, use the command `M-x gdb` in Emacs. Give the executable file you want to debug as an argument. This command starts GDB as a subprocess of Emacs, with input and output through a newly created Emacs buffer.

Using GDB under Emacs is just like using GDB normally, except all “terminal” input and output goes through the Emacs buffer. This applies both to GDB

commands and their output, and to the input and output done by the program you are debugging. This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way. All the facilities of Emacs' Shell mode are available for interacting with your program. In particular, you can send signals the usual way—for example, **Ctrl-c**, **Ctrl-c** for an interrupt, **Ctrl-c**, **Ctrl-z** for a stop. GDB displays source code through Emacs.

Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (`=>`) at the left margin of the current line. Emacs uses a separate buffer for source display, and splits the screen to show both your GDB session and the source.

Explicit GDB `list` or `search` commands still produce output as usual, but you probably have no reason to use them from Emacs.



CAUTION! If the directory where your program resides is not your current directory, it can be easy to confuse Emacs about the location of the source files, in which case the auxiliary display buffer does not appear to show your source.

GDB can find programs by searching your environment's `PATH` variable, so the GDB input and output session proceeds normally; but Emacs does not get enough information back from GDB to locate the source files in this situation.

To avoid this problem, either start GDB mode from the directory where your program resides, or specify an absolute file name when prompted for the `M-x gdb` argument.

A similar confusion can result if you use the GDB file command to switch to debugging a program in some other location, from an existing GDB buffer in Emacs.

By default, using the keystroke sequence, `M-x gdb` calls the program called `gdb`. If you need to call GDB by a different name (for example, if you keep several configurations around, with different names) you can set the Emacs variable `gdb-command-name`.

For example, `(setq gdb-command-name "mygdb")`—which is preceded by using the keystroke sequence, **Esc**, **Esc**, or typed in the `*scratch*` buffer, or in your `.emacs` file—makes Emacs call the `"mygdb"` program instead.

In the GDB I/O buffer, you can use these special keystroke sequences of Emacs commands in addition to the standard Shell mode commands in Table 2-4.

Table 2-4: Shell Mode Commands

Command	Description
C-h, m	Describe the features of Emacs' GDB Mode.
M-s	Execute to another source line, like the GDB <code>step</code> command; also update the display window to show the current file and location.
M-n	Execute to next source line in this function, skipping all function calls, like the GDB <code>next</code> command. Then update the display window to show the current file and location.
M-i	Execute one instruction, like the GDB <code>stepi</code> command; update display window accordingly.
M-x, gdb-nexti	Execute to next instruction, using the GDB <code>nexti</code> command; update display window accordingly.
Ctrl-c, Ctrl-f	Execute until exit from the selected stack frame, like the GDB <code>finish</code> command.
M-c	Continue execution of your program, like the GDB <code>continue</code> command. Note: In Emacs version 19, this command uses the Ctrl-c, Ctrl-p keystroke sequence.
M-u	Go up the number of frames indicated by the numeric argument (see "Numeric Arguments" in <i>The GNU Emacs Manual</i>), like the GDB <code>up</code> command. Note: In Emacs version 19, this command uses the Ctrl-c, Ctrl-u keystroke sequence.

Table 2-4: Shell Mode Commands(Continued)

Command	Description
M-d	Go down the number of frames indicated by the numeric argument, like the GDB <code>down</code> command. Note: In Emacs version 19, this command uses the Ctrl-c, Ctrl-d keystroke sequence.
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 <code>disassemble</code> by using the Ctrl-x, & keystroke sequence. You can customize this further by defining elements of the list <code>gdb-print-command</code> ; once it is defined, you can format or otherwise process numbers picked up by using the Ctrl-x, & keystroke sequence 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, the Emacs command using the **C-x, Spacebar** keystroke sequence 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 type the command, **f**, in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers.

If you add or delete lines from the text, the line numbers that GDB knows cease to correspond properly with the code.

Command Line Editing

The following material describes GNU's command line editing interface.

Introduction to Line Editing

The following paragraphs describe the notation we use to represent keystrokes.

NOTE: The text **Ctrl-K** is read as “Control K” and describes the command to produce when using the **Control** and the **K** keys sequence. The text **M-K** is read as “Meta K” and describes the command to produce when using the **meta** key (if you have one, it may be the key with a diamond), and the **K** key. If you do not have a **meta** key, the identical keystroke can be generated by using the **Esc** key and then **K**. Either process is known as “meta-fying the K key.” The text **M-Ctrl-K** is read as “Meta Control K” and describes the command to produce when asked to “meta-fy C K.”

NOTE: The hyphen characters and the comma characters are not a part of the keystroke sequence to type.

All uppercase letters require using the shift key, of course, since all commands are case sensitive.

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 127 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.

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 **Delete** to back up, and 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 **Ctrl-B** to move the cursor to the left, and then correct your mistake. Afterward, you can move the cursor to the right with **Ctrl-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 essentials for editing the text of an input line are in the following table.

Command	Action
Ctrl-B	Move back one character.
Ctrl-F	Move forward one character.
Delete	Delete the character to the left of the cursor.
Ctrl-D	Delete the character underneath the cursor.
Printing characters	Insert itself into the line at the cursor.
Ctrl-_	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. For your convenience, many other commands have been added in addition to **Ctrl-B**, **Ctrl-F**, **Ctrl-D**, and **Delete**.

Here are some commands for moving more rapidly about the line:

Command	Action
Ctrl-A	Move to the start of the line.
Ctrl-E	Move to the end of the line.
M-F	Move forward a word.
M-B	Move backward a word.
Ctrl-L	Clear the screen, reprinting the current line at the top.

Notice how **Ctrl-F** moves forward a character, while **M-F** moves forward a word. It is a loose convention 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. Table 2-8 is a list of commands for killing text.

Command	Action
Ctrl-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	Kill from the cursor to the start of the previous word, or if between words, to the start of the previous word.
Ctrl-W	Kill from the cursor to the previous whitespace.
Ctrl-L	Clear the screen, reprinting the current line at the top. This is different than M-Delete because the word boundaries differ.

The following table shows how to yank the text back into the line.

Command	Action
Ctrl-Y	Yank the most recently killed text back into the buffer at the cursor. 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 Ctrl-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 arguments act 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-- Ctrl-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 **Ctrl-D** command an argument of 10, you could use the keystroke sequence, **M-1, 0, Ctrl-D**.

Readline Init File

Although the Readline library comes with a set of GNU Emacs-like keybindings, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an *init* file in your home directory. The name of this file is `~/.inputrc`.

When a program which uses the Readline library starts up, the `~/.inputrc` file is read, and the keybindings are set.

In addition, the **Ctrl-X, Ctrl-R** command re-reads this init file, thus incorporating any changes that you might have made to it.

Readline Init Syntax

There are only four constructs allowed in the `~/.inputrc` file.

Variable Settings

You can change the state of a few variables in Readline. You do this by using the `set` command within the init file. Here is how you would specify that you wish to use `vi` line editing commands:

```
set editing-mode vi
```

Right now, there are only a few variables which can be set; so few, in fact, that we just iterate them here:

```
editing-mode
```

The `editing-mode` variable controls which editing mode you are using. By default, GNU Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can either be set to `emacs` or `vi`.

```
horizontal-scroll-mode
```

This variable can either be set to `On` or `Off`. Setting it to `On` means that the text of the lines that you edit will scroll horizontally on a single screen line when they are larger than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to `Off`.

```
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.

```
prefer-visible-bell
```

If this `Off` variable is set to `On` it means to use a visible bell if one is available, rather than simply ringing the terminal bell. By default, the value is `Off`.

Key Bindings

The syntax for controlling keybindings in the `~/.inputrc` file is simple. First you have to know the name of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

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: *function-name* or *macro*

keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: "&output"
```

In the example, **Ctrl-U** is bound to the function, `universal-argument`, and **Ctrl-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. Simply place the key sequence in double quotes.

GNU Emacs style key escapes can be used, as in the following example:

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the example, **Ctrl-U** is bound to the function `universal-argument` (just as it was in the first example), **Ctrl-X**, **Ctrl-R** is bound to the function `reread-init-file`, and **Esc-[, 1, 1, ~** is bound to insert the text `Function Key 1`. See the following table for additional information.

Command	Action
beginning-of-line (Ctrl-A)	Move to the start of the current line.
end-of-line (Ctrl-E)	Move to the end of the line.
forward-char (Ctrl-F)	Move forward a character.
backward-char (Ctrl-B)	Move back a character.

Command	Action
forward-word (M-F)	Move forward to end of the next word.
backward-word (M-B)	Move back to the start of this, or the previous, word.
clear-screen (Ctrl-L)	Clear the screen leaving the current line at the top of the screen.

Command	Action
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 (Ctrl-P)	Move 'up' through the history list.
next-history (Ctrl-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 (Ctrl-R)	Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.
forward-search-history (Ctrl-S)	Search forward starting at the current line and moving 'down' through the the history as necessary.

Command	Action
delete-char (Ctrl-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 Ctrl-D , then return EOF.
backward-delete-char (Rubout)	Delete the character behind the cursor. A numeric argument says to kill the characters instead of deleting them.
quoted-insert (Ctrl-Q, Ctrl-V)	Add the next character that you type to the line verbatim. This is how to insert things like Ctrl-Q , for example
tab-insert (M-Tab)	Insert a tab character.

Command	Action
self-insert (a, b, A, 1, !, ...)	Insert yourself.
transpose-chars (Ctrl-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.

Command	Action
<code>downcase-word (M-L)</code>	Lowercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.
<code>capitalize-word (M-C)</code>	Uppercase the first letter in the current (or following) word. With a negative argument, do the previous word, but do not move point.
<code>kill-line (Ctrl-K)</code>	Kill the text from the current cursor position to the end of the line.
<code>backward-kill-line ()</code>	Kill backward to the beginning of the line. This is normally unbound.
<code>kill-word (M-D)</code>	Kill from the cursor to the end of the current word, or if between words, to the end of the next word.
<code>backward-kill-word (M-Delete)</code>	Kill the word behind the cursor.
<code>unix-line-discard (Ctrl-U)</code>	Kill the whole line the way Ctrl-U used to in UNIX line input. The killed text is saved on the kill-ring.
<code>unix-word-rubout (Ctrl-W)</code>	Kill the word the way Ctrl-W used to in UNIX line input. The killed text is saved on the kill-ring. This is different than <code>backward-kill-word</code> because the word boundaries differ.
<code>yank (Ctrl-Y)</code>	Yank the top of the kill ring into the buffer at point.
<code>yank-pop (M-Y)</code>	Rotate the kill-ring, and yank the new top. You can only do this if the prior command is <code>yank</code> or <code>yank-pop</code> .

Command	Action
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 ()	Do what Ctrl-U does in GNU Emacs. By default, this is not bound.

Command	Action
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.

Command	Action
<code>reread-init-file</code> (Ctrl-X, Ctrl-R)	Read in the contents of your <code>~/.inputrc</code> file, and incorporate any bindings found there.
<code>abort</code> (Ctrl-G)	Stop running the current editing command.
<code>prefix-meta</code> (Esc)	Make the next character that you type be metafied. This is for people without a meta key. Typing ESC F is equivalent to typing M-F .
<code>undo</code> (Ctrl-_)	Incremental undo, separately remembered for each line.
<code>revert-line</code> (M-R)	Undo all changes made to this line. This is like typing <code>undo</code> enough times to get back to the beginning.

Readline 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 GNU Emacs and `vi` editing modes, use the command **M-Ctrl-J** (`toggle-editing-mode`). When you enter a line in `vi` mode, you are already placed in `insertion` mode, as if you had typed an `i`. 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`, and following lines with `j`, and so forth.

Using History Interactively

The following describes how to use the GNU History Library interactively, from a user's standpoint.

History Interaction

The History library provides a history expansion feature similar to the history expansion in `csh`. The following text describes the syntax you use to manipulate history information.

History expansion takes two parts. In the first part, determine which line from the previous history will be used for substitution. This line is called the event. In the second part, select portions of that line for inclusion into the current line. These portions are called words. 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.

Event Designators

An event designator is a reference to a command line entry in the history list.

<code>!</code>	Start a history substitution, except when followed by a space, tab, or the end of the line... = or (.
<code>!!</code>	Refer to the previous command. This is a synonym for <code>!-1</code> .
<code>!n</code>	Refer to command line <i>n</i> .
<code>!-n</code>	Refer to the command line <i>n</i> lines back.
<code>!string</code>	Refer to the most recent command starting with <i>string</i> .
<code>!?string[?]</code>	Refer to the most recent command containing <i>string</i> .

Word Designators

`A:` separates the event designator from the word designator. It can be omitted if the word designator begins with a `^`, `$`, `*` or `%`. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

<code>0 (zero)</code>	The zero'th word. For many applications, this is the command word.
<code>n</code>	The <i>n</i> 'th word.

<code>^</code>	The first argument. that is, word 1.
<code>T\$</code>	The last argument.
<code>%</code>	The word matched by the most recent <code>?string?</code> search.
<code>x-y</code>	A range of words; <code>-y</code> abbreviates <code>0-y</code> .
<code>*</code>	All of the words, excepting the zero'th. This is a synonym for <code>1-\$</code> . It is not an error to use <code>*</code> 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 `:`.

<code>#</code>	The entire command line typed so far. This means the current command, not the previous command.
<code>h</code>	Remove a trailing pathname component, leaving only the head.
<code>r</code>	Remove a trailing suffix of the form <code>'.'</code> <i>suffix</i> , leaving the basename.
<code>e</code>	Remove all but the suffix.
<code>t</code>	Remove all leading pathname components, leaving the tail.
<code>p</code>	Print the new command but do not execute it.

CHAPTER 3 *LynxOS GDB Enhancements*

LynxOS GDB extends and enhances the functionality of the GNU debugger for debugging various LynxOS targets. This chapter is intended to supplement the previous chapter, See “Debugging with GDB” on page 11. Readers are advised to read the chapter prior to this in order to familiarize themselves with GDB.

Overview

LynxOS GDB supports debugging of a variety of LynxOS targets including, but not limited to, the following areas:

- POSIX threads
- Remote applications
- Shared libraries
- LynxOS kernel device drivers.

This chapter shows command prompts as follows:

A command entry to GDB prompt (on the host):

```
(gdb) example command
```

A command entry to the host’s shell prompt:

```
myhost$> example command
```

A command entry to the target’s shell prompt:

```
mytarget$> example command
```

Debugging POSIX Threads

LynxOS user threads fully conform to the POSIX/IEEE 1003.1c threads model. A LynxOS process consists of one or more threads each of which is scheduled by the kernel. LynxOS GDB provides full support of multiple thread debugging including

- Browsing threads in a process
- Switching focus among threads
- Setting breakpoints either common to all threads in a process or specific to a particular thread.

For more detailed information, see “Debugging Programs with Multiple Threads” on page 34 in the previous chapter.

NOTE: GDB can debug a single process per debug session. To debug more than one process, start as many GDB sessions as the number of the processes. Each GDB session works independently from the others and there is no mutual synchronization mechanism available between GDB sessions.

Understanding Thread Numbers

Each debugged process may contain any number of threads. GDB manages these threads with internal thread numbers that are unique within the *process*, and within the GDB session. Note that these thread numbers are different from LynxOS thread IDs that are assigned by the LynxOS kernel and are unique throughout the *operating system*. GDB maintains the mapping between its thread numbers and LynxOS thread IDs.

Browsing and Switching Threads

To browse all the threads in the process, use the `info thread` command:

```
(gdb) info thread
* 1 process 8 thread 38 0x100029d4 in _trap_ ()
  2 process 8 thread 32 0x10002dd0 in _trap_ ()
(gdb)
```

The first column of a thread list is the GDB thread number. The asterisk indicates the current thread. The second and third numbers (8, 38, and 32 in this example) are the LynxOS process ID and thread IDs, respectively.

Current Thread

Whenever GDB stops and returns to its prompt, it maintains its concept of the *current thread*. GDB can only focus on one thread at a time, which is referred to as the current thread. By default, any GDB command uses the current thread if it implicitly uses thread-specific parameters such as

- Expressions that contain an automatic variable
- Browsing registers
- Browsing the call stack chain.

At the start-up of the debugged process, the initial thread is the current thread. When a thread hits a breakpoint or watchpoint, that thread becomes the current thread. When the target process is interrupted by GDB, then the interrupted thread becomes the current thread.

Use the `thread` command with the new thread number to switch the focus from one thread to another:

```
(gdb) thread 2
```

Setting a Breakpoint

You can make a breakpoint common to all the threads, so that any thread in the target process will stop at a hit on it, or specific to a particular thread, so that only the specified thread will stop. The default is any thread.

To set a thread-specific breakpoint, use the `break` command with the `thread` modifier:

```
(gdb) break foo.c:123 thread 2
```

When a thread stops because of a breakpoint or any other reason, all threads in the target process will stop immediately, not just the thread that encountered the stop condition. Likewise, when a thread is resumed, all threads in the debug process are resumed. Whenever GDB is at its prompt, the entire process is stopped and all variables can be determined statically.

NOTE: Thread-specific breakpoints are implemented via simulation: All breakpoints are actually thread-insensitive to the operating system. When GDB detects a hit at a thread-specific breakpoint by a thread that is not specified for the breakpoint, GDB immediately resumes the target without reporting it to the user. So, the operation is transparent, but some speed penalty may be seen if there are many uninteresting hits.

Resuming Threads

The `continue` command resumes the operation of the target process. If the target process has more than one thread, all the threads are resumed, not just the current thread.

Likewise, single-stepping actually resumes all the threads in the target process. The `step` and `stepi` commands merely guarantee that the current thread executes at most one line of code and one instruction respectively, while other threads with the same or higher priorities than the current thread may execute any amount of code before control returns to GDB. It is even possible that the current thread may not have a chance to complete single-stepping if some other thread runs first and hits a breakpoint or receives a signal.

To single-step the current thread, raise the thread's priority to a value higher than any other thread in the target process by using the `setprio` command at the shell level.

You can run `setprio` (process 8, thread 38) at the target's shell prompt:

```
mytarget$> setprio 20 8.38
```

or if the target is a remote machine:

```
(gdb) rshell setprio 20 8.38
```

The thread's original priority must be restored after completing the exclusive single-stepping for normal operation of the application.

NOTE: When a multithreaded target process is resumed, the threads are actually resumed one after another by the LynxOS kernel, not all at once. The order of resumption and subsequent execution of threads is determined by the scheduling algorithm of the LynxOS kernel based on priorities.

Debugging Embedded Applications Remotely

GDB can be used to debug embedded applications remotely in the following ways:

- Debugging user processes with full support of signal, process attach, and so on
- Debugging device drivers (kernel) over a serial line

For more information, see “Commands for Managing Targets” on page 113 and “Using the gdbserver program” on page 115.

In addition, LynxOS GDB supports the following features:

- Remote start of `gdbserver` from GDB
- Extension of serial line communication to TCP/IP using a proxy.

Using the Target Command

Use the `target` command to choose the appropriate remote debug target and protocol (communication channel). In the following syntax, the first argument specifies the remote debug target and the second specifies the protocol.

```
(gdb) target debug-target protocol [args]
```

LynxOS GDB supports the remote debug targets shown in Table 3-1:

Table 3-1: Remote Debug Targets

Target	Target name(s)
Remote user process	<code>remote</code> and <code>extended-remote</code>
Kernel/device driver	<code>skdb</code>

LynxOS GDB supports the remote debug protocols shown in Table 3-2.

Table 3-2: Remote Debug Protocols

Protocol	Example
TCP/IP	<code>foo:12345</code>
Serial line	<code>/dev/ttya</code>
Remotely started <code>gdbserver</code>	<code>foo:gdbserver</code>
Proxy server	<code>foo:sspp</code>

The optional third argument `args` to the `target` command is only used by the remote `gdbserver` and proxy server protocols. Refer to the following sections for the details of `args`.

Debugging Remote Targets

Remote and Extended-Remote Targets

Both `remote` and `extended-remote` targets select debugging of a remote user process over a communication channel. There are small differences between `remote` and `extended-remote` targets in the way `gdbserver` handles termination of a debug session as shown in Table 3-3.

Table 3-3: Remote and Extended-Remote Targets

Target	Kill Command	Target Process Exits or is Killed By Signal	Communication Error or New Target is Selected
<code>remote</code>	<code>gdbserver</code> exits	<code>gdbserver</code> exits	<code>gdbserver</code> reopens communication
<code>extended-remote</code>	<code>gdbserver</code> respawns target process	<code>gdbserver</code> respawns target process	<code>gdbserver</code> exits

`remote` and `extended-remote` targets support a full range of GDB features available in LynxOS GDB, including debugging of multithreaded processes, sending a signal to the target process, and attaching to a target process.

Device Driver/Kernel Target (`skdb`)

LynxOS GDB supports debugging of kernel code including device drivers over a serial line communication. For more information, see “Debugging Kernel/Device Drivers” on page 158.

Supported Protocols for Remote and Extended-Remote Targets

TCP Port

If there is TCP/IP communication available between the debugging host and the remote debugging target, it can be used to get the best possible debugging speed and reliability.

To start remote debugging through a TCP port, the remote target must first start `gdbserver` with an unused TCP port number that is available for normal use.

The next example command lines are typed in to the target machine:

```
mytarget$> gdbserver junk:12345 /test/prog arg
```

or

```
mytarget$> gdbserver foo:23456
```

The first example starts the target program `/test/prog` with an argument `arg`. The second example does not start a target program immediately but waits until it is told by the host GDB to attach an already-running process. For more information, see “Debugging Kernel/Device Drivers” on page 158.

In either case, the first argument to `gdbserver` is the TCP port specification in the form of `host:port`. An unused TCP/IP port number must be specified for `gdb`. This is usually a large number between 1,024 and 65,534 inclusive; a range of 5,000 through 65,534 is recommended for better compatibility. The host name string before `:` can be anything and is ignored by `gdbserver`.

At the host machine, give the host GDB prompt the same port number associated with the target’s host name or IP address:

```
(gdb) target remote mytarget:12345
```

or

```
(gdb) target extended-remote 198.4.254.217:23456
```

Using a Serial Line

A serial communication line can be used for GDB remote debugging with `gdbserver` provided that a serial port is available on both the host and target machines. When using a serial port, make sure that no other processes are using the port on either side. Neither GDB nor `gdbserver` uses a lock file, but they must gain exclusive access to the ports; otherwise, GDB may report a communication error.

To start `gdbserver` with the target process on the target’s serial port (`/dev/com1`):

```
mytarget$> gdbserver /dev/com1 /test/prog arg
```

The following starts `gdbserver` for later process attaching on the target’s serial port (`/dev/com2`). The `-b` option is used to specify the serial port’s communication speed explicitly.

```
mytarget$> gdbserver -b 19200 /dev/com2
```

In either case, `/dev/com1` and `/dev/com2` are the device files of the targets' serial ports that are connected to the GDB hosts.

At the host machine, you must give the GDB prompt the *host's* serial port name (`/dev/ttya` in this case) to which the target is connected:

```
(gdb) target remote /dev/ttya
```

If the GDB host is Windows, it resembles the following:

```
(gdb) target remote com2
```

To change the host's serial line speed, use the `set remotebaud` command *before* using the `target` command:

```
(gdb) set remotebaud 19200
```

Starting gdbserver Remotely

With TCP/IP communication, you can start `gdbserver` remotely in a couple of different ways:

- Through a `telnet` session on the target from the GDB host
- From the target's start-up script
- With the `rsh` command from the debug host
- From GDB.

The last method has an advantage that, for example, you can start a debug session completely from GDB, but it, as well as the `rsh` method, requires that the remote shell (`rsh`) and the `.rhosts` file for your account on the target be set up so that the GDB user can use remote shell commands on the target from the GDB host.

```
(gdb) target remote mytarget: /test/prog arg
```

or

```
(gdb) target remote mytarget:gdbserver
```

The first example starts `gdbserver` on the remote target with the target program `/test/prog` and an argument of `arg`. The second starts `gdbserver` for later attachment of a target process. The `gdbserver` string after the colon ":" is optional; if it is present, it must be in all lower-case and cannot be preceded by a path prefix.

In both of the above cases, the host GDB process spawns a local `rsh` process to ask the remote `rshd` process to start `gdbserver` with appropriate arguments. The communication TCP port number is automatically determined and you do not need to specify it. The standard output and standard error paths of the remote target process are redirected to those of the local GDB process, while the standard input path of the remote target process is either closed or redirected to the local host's `/dev/null`. Therefore, an interactive program cannot be debugged in this way.

Using a Proxy Server

The serial line between the GDB host and the target usually limits the physical distance between the two machines. It is, however, often desirable to be able to debug a target that has only a serial line for communication from a geographically distant locale. LynxOS GDB provides a proxy server solution to this problem.

The proxy server program runs on a third computer that is actually connected to the target via a serial line. The proxy computer and the GDB host computer communicate over TCP/IP and the proxy server program redirects all the messages to and from the serial line to the TCP/IP connection. This way, one can debug the target from a local workstation placed anywhere on the globe as long as there is a TCP/IP connection with the proxy server.

To enable remote debugging with a proxy, start `gdbserver` on the target with the serial port name connected to the proxy computer first:

```
mytarget$> gdbserver /dev/com2 /test/prog arg
```

Then specify the target `sspp` at the GDB prompt:

```
(gdb) target myproxy:sspp /dev/ttya
```

Here, `myproxy` is the proxy server's host name, which could be a fully qualified domain name (FQDN) such as `myproxy.foo.com` or an IP address such as `198.4.254.47`; and `/dev/ttya` is the serial port name of the proxy computer to which the target machine is connected.

This proxy server is also useful when the target's serial port is connected to a "serial terminal server" computer. The terminal server can be configured to run the proxy for shared access to the target. For more details on the proxy server, see "Proxy Server" on page 165.

Starting the Remote Target

There are two ways to start the remote target process:

- Start from `gdbserver`
- Attach to a Running Process

The next sections describe these ways of starting the remote target.

Starting from `gdbserver`

To start the remote target process from `gdbserver`, give the target program name with optional arguments at the `gdbserver` command line:

```
mytarget$> gdbserver junk:12345 /test/prog arg
(gdb) target remote mytarget:12345
```

or

```
mytarget$> gdbserver /dev/com2 /test/prog arg
(gdb) target remote /dev/ttya
```

or

```
(gdb) target remote mytarget: /test/prog arg
```

The first and second examples start `gdbserver` on the remote target then make a connection to the `gdbserver` process from GDB. The third example starts `gdbserver` remotely from GDB over a TCP/IP connection.

In any case, as soon as the communication is established, one will see something similar to the following:

```
(gdb) target remote mytarget: /test/prog arg
Process /test/proc created; pid = 17
Connected to 198.4.254.217...
Remote debugging using 198.4.254.217: /test/prog arg
Kernel supports MTD ptrace requests.
gdbserver: passed 0.0.0.0:3304 using addr:port=198.4.254.131:3304
0x10001000 in __start ()
(gdb)
```

At this point, the target process has started and is stopped at the very first user instruction of the program, which is in this case labeled as `__start`. To let the target process run, use the `continue` command (after setting breakpoints or other desired debugger commands), but do not use the `run` command which would start a local process.

Attaching to a Running Process

In a multiprocess application where the target process is forked by another process on the target, it is not possible for `gdbserver` to start the target process with the target program specification given in the GDB command line. In such a case, GDB can tell `gdbserver` to attach to an already-running process.

To start `gdbserver` for attaching to a process, give no target program specification to it:

```
mytarget$> gdbserver junk:12345
(gdb) target remote mytarget:12345
```

or

```
mytarget$> gdbserver /dev/com2
(gdb) target remote /dev/ttya
```

or

```
(gdb) target remote mytarget:
```

The first and second examples start `gdbserver` on the remote target then make a connection to the `gdbserver` process from GDB. The third example starts `gdbserver` remotely from GDB over a TCP/IP connection.

To attach to a remote process, one needs to obtain the target process ID. To find out the process ID, execute the `ps` command on the target. With LynxOS GDB, you can run the `ps` command remotely from your GDB host with the `rshell` GDB command.

Then, use the GDB attach command with the process ID to attach the target process:

```
(gdb) target remote mytarget:

Ready to attach to a process
gdbserver: passed 0.0.0.0:1050 using addr:port=198.4.254.47:1050
Connected to mytarget...
Remote debugging using mytarget:

(gdb) rshell ps

pid  ppid  grp  pri  text  stk  data  time  dev  user  S  name
 34   32   34   17   336  32   132  0.58  tty0  joe  W  /bin/bash
 36   13   36   17   336* 32   132  0.46  tty1  joe  W  /bin/bash
 44   34   44   17   28   8    8    0.01  tty0  joe  W  /test/prog
6596K/0K free physical/virtual, 708K used (in this display)

(gdb) attach 44

Attaching to remote program `/usr/home/joe/test/prog' (process 44)...
Kernel supports MTD ptrace requests.
```

```
Process 44 has threads 39.  
0x100029d4 in _trap_ (  
  
(gdb)
```

To resume the target, use the `continue` command (after setting breakpoints, if necessary).

Target's Environment

Although GDB supports `set environment` to set the debug target's environment, currently this command has no effect on remote debugging. To set or reset the remote target process's environment, do so on the parent process (usually the shell) of `gdbserver` (if the application process is started by `gdbserver`) or the application process (if one attaches to the process later).

Postmortem Debugging of Dynamically Linked Programs

When loading a partial core file without either the data or the heap section created from a dynamically linked program, GDB is unable to debug the shared libraries used by the program. To obtain the shared libraries address for correct interpreting of the libraries symbolic information, GDB uses the `DT_DEBUG` symbol. The `DT_DEBUG` pointer maintained by the dynamic linker is located in the program data section of the core file and points to a special program heap area. This program heap area is allocated by the dynamic linker and holds the structures containing the information on how the dynamic linker loads the libraries needed for the program execution. In the case when the core file is configured not to contain either the data or the program heap section, GDB is unable to get the load information of the library and analyze the functions located in the shared libraries.

Should debugging of the shared libraries in a core file be required, the user must configure the core file to include the data and heap sections. Refer to the *LynxOS User's Guide* for details.

Debugging Shared Libraries

A shared library is a collection of library functions that are commonly used by multiple application programs at the same time. Instead of linking a library to each application program executable, a single copy of the shared library is loaded into memory and used by multiple programs. This sharing reduces the use of physical

memory as well as disk storage requirements; therefore, it is especially useful for embedded applications where memory is tight.

LynxOS GDB can set breakpoints in, single-step, and trace shared library code just like any application program. When a breakpoint is set in the text segment of a shared library that is actually used by multiple processes, a copy of the page where the breakpoint is being set is made for the debugged process so that the breakpoint does not interfere with other processes that share the same shared library.

Creating a Shared Library for Debugging Purposes

To create a shared library for GDB debugging purposes, compile (`gcc`) the library source file(s) with the `-g` option.

Loading Shared Library Symbol Information

GDB automatically loads the necessary symbol files (shared library files) when one of the following GDB commands is executed:

- `run`
- `attach`
- `target (extended-)remote`
- `target core` (or the `-c` option or a core file is given in the GDB command arguments—postmortem debugging)

The following `add-symbol-file` command can be used to load a shared library symbol file. This may be necessary if the file cannot be found in one of the automatically searched directory paths, or if you are performing postmortem (core file) debugging:

```
(gdb) add-symbol-file ./libmy.so 0
```

The first argument for the `add-symbol-file` command is the additional shared library file path name on the debug host's file system.

The second argument (0) is a dummy argument, and GDB ignores it.

Application-Loaded (dlopen'ed) Shared Libraries

GDB automatically loads the necessary symbol file (shared library file) when an application-loaded shared library file is first opened by the application (`dlopen()`) and discards the symbol file when the shared library file is last closed by the

application (`dlopen()`) for live debugging. The `dlopen` and `dclose` library functions use special signals to notify GDB that a shared library has been opened or closed.

GDB also searches for application-loaded shared libraries and loads the necessary symbol files when it attaches to a live target process or when it starts analyzing a core file (postmortem debugging).

The following `add-symbol-file` command can be used to load a shared library symbol file manually. This may be necessary if GDB cannot locate the file in any of the automatically searched directory paths:

```
(gdb) add-symbol-file ./shlib2.so 0
```

The first argument for the `add-symbol-file` command is the additional shared library file path name on the debug host's file system.

The second argument (0) is a dummy argument, and GDB ignores it.

If a symbol file has been loaded manually with `add-symbol-file`, it must be discarded manually with the `delete-symbol-file` command after the shared library has been last closed by the `dclose()` library call. GDB will prompt the user when this seems to be necessary.

The `delete-symbol-file` command takes one argument for the deleted symbol file name on the debug host's file system. The `info symbol-file` command shows the list of currently loaded symbol files.

```
(gdb) info symbol-file
```

From	To	Symbol file
0x00400a20	0x00400e1f	/usr/lynx/3.1.0/mips/tmp/hello/hello
0x70000c20	0x7000450f	/usr/lynx/3.1.0/mips/lib/shlib/libdl.so
0x70027f40	0x7005987f	/usr/lynx/3.1.0/mips/lib/shlib/libc.so
0x70452ec0	0x704649ff	/usr/lynx/3.1.0/mips/lib/shlib/libgcc.so
0x70475240	0x70480aef	/usr/lynx/3.1.0/mips/lib/shlib/libm.so

```
(gdb) delete-symbolfile \  
/usr/lynx/3.1.0/mips/lib/shlib/libdl.so
```

Deferred Breakpoints

When setting a breakpoint with symbolic information such as a function name, GDB has to resolve the breakpoint specification into the target process's virtual address immediately. This is impossible if the breakpoint being set will be found in a shared library and the shared library symbol file has not been loaded to GDB either because the target process has not yet started or because GDB has not yet attached to the target process.

GDB for LynxOS supports deferred breakpoints that allow breakpoint addresses to remain unresolved. When a deferred breakpoint is set, GDB immediately tries to set it as a “real” breakpoint: If the breakpoint setting is successful, the breakpoint will work as a regular breakpoint; if it fails, GDB does not print an error message but will remember the breakpoint specification by keeping it in the deferred breakpoint list. When the target process starts or is attached to GDB and a shared library is detected and loaded, GDB will try to set the deferred breakpoints as real breakpoints.

Deferred Breakpoint Commands

The following GDB commands are available for supporting deferred breakpoints:

```
(gdb) dbreak breakpoint_spec
```

The `dbreak` command sets a deferred breakpoint. *breakpoint_spec* is the specification of the breakpoint to be set; it can be any string that would be accepted by the “real” `break` command, including an optional `if` condition clause. If the breakpoint is successfully set as a real breakpoint, the breakpoint will work just like any other regular breakpoints, except that it is also registered in the deferred breakpoint list; otherwise, *breakpoint_spec* merely remains registered in the deferred breakpoint list. An attempt will be made to set the deferred breakpoint as a real breakpoint when the target process is started or attached by one of the following commands:

- `run`
- `attach`
- `target (extended-)remote`

If the attempt to convert a deferred breakpoint to a real breakpoint fails, the deferred breakpoint will remain registered for another attempt in the future and you will see no error message.

NOTE: The `dbreak` command does not parse or check the syntax of *breakpoint_spec*; it simply passes the whole string to the breakpoint command executive. So, “failure to set as a real breakpoint” may indicate a syntactical error in *breakpoint_spec*.

Optionally, you may give the `break` command an unresolvable breakpoint specification. If the `break` command finds it cannot resolve the breakpoint specification to an address immediately, it will fall into the `dbreak` command function after confirmation.

```
(gdb) break dlfunc
```

```
Function "dlfunc" not defined. Make deferred? (y or n) y  
Deferred breakpoint set for "dlfunc."
```

```
(gdb) ddelete [dbreak_num]
```

The `ddelete` command removes a deferred breakpoint from the deferred breakpoint list. `dbreak_num` is the deferred breakpoint number shown by the `info dbreak` command. If `dbreak_num` is not given, `ddelete` will remove all deferred breakpoints after confirmation. If the deferred breakpoint being removed is currently set as a real breakpoint, `ddelete` will prompt for confirmation to remove the real breakpoint as well.

NOTE: The deferred breakpoint list uses separate numbering from “real breakpoints.” Those two series of breakpoint numbers should not be confused.

```
(gdb) info dbreak [dbreak_num]
```

The `info dbreak` command displays the information about the deferred breakpoint designated by `dbreak_num` or all deferred breakpoints.

Deferred Breakpoints for Application-Loaded (dlopen'ed) Shared Libraries

In addition to the above features of deferred breakpoints that are applicable to both kernel-loaded and application-loaded shared libraries, application-loaded (dlopen'ed) shared libraries benefit some more from deferred breakpoints.

Automatic Promotion for dlopen

When a new application-loaded (dlopen'ed) shared library is loaded by the target process and its symbol file is loaded to GDB, GDB will automatically try to set the “pending” deferred breakpoints as real breakpoints. Those deferred breakpoints that are successfully converted to real breakpoints are marked “busy,” while other pending ones will remain pending for later attempts. GDB will not remove the successfully converted deferred breakpoints from the list; they will remain registered until explicitly removed by the `ddelete` command. This is because those breakpoints may need to be set again when the shared library is closed and then reopened in the future.

Automatic Demotion for dlclose

When an application-loaded shared library is last closed and its symbol file is removed from GDB, GDB automatically removes all breakpoints that were set for the shared library. These breakpoints include both those that were automatically set “real” when the shared library was loaded and those that were set manually by the `break` command.

It is important to know that a stale breakpoint is *deleted* but *not disabled* even if the breakpoint belonged to an application-loaded shared library that may be reopened in the future. This is because the breakpoint’s address value is no longer valid and there is no guarantee that the same address will be used for the given deferred breakpoint specification when the shared library is reopened. Therefore, the corresponding real breakpoint number will become invalid as well, and a new breakpoint number will be assigned when the shared library is reopened.

Shared Library File Path Names

In remote debugging, the debug host and the debug target may not necessarily have the same directory layout; for example, one may be developing a shared library `foo.so` in the debug host’s directory `/home/joe/proj1/usr/lib/shlib/`, but the library may be supposed to be loaded by the target process from `/usr/lib/shlib/` on the debug target’s file system. GDB has to resolve this directory path difference particularly for shared library symbol file loading because:

- For automatic shared library symbol loading, the shared library file path names are extracted from the application executable in the target’s notation (`/usr/lib/shlib/foo.so` in the above example), while GDB must load the symbol file from the debug host’s file system (for example: `/home/joe/proj1/usr/lib/shlib/foo.so`)
- For manual shared library symbol loading (`add-symbol-file`), the shared library file names are given in the debug *host’s* notation. GDB has to translate it into the debug target’s notation in order to obtain the dynamic loading address information from the target.

In general, it is recommended to have (a subset of) the target’s file system image under a host file system directory pointed to by the `ENV_PREFIX` environment variable, but this may not always be the case. LynxOS GDB handles this shared library file path resolution issue in the following ways for convenience and flexibility.

Automatic Shared Library Symbol File Loading

The `.dynamic` section of the application's executable file will contain the shared library file names and optionally the directory information (on the target's file system). GDB uses the `_host_shlib_dirs` GDB variable and the debug host's `ENV_PREFIX` environment variable to search for the shared library symbol files on the debug host's file system.

`_host_shlib_dirs` is a colon-separated list of *host* directory names in which the shared library file is expected to reside. GDB uses these directories to override the following file name path composition rules. `_host_shlib_dirs` can be set by the `set` command as follows:

```
(gdb) set _host_shlib_dirs
/home/joe/proj1/usr/lib/shlib:/home/joe/proj1/testlib
```

The above example implies the shared library symbol files are supposed to be found in `/home/joe/proj1/usr/lib/shlib` or `/home/joe/proj1/testlib`.

Otherwise if `_host_shlib_dirs` is not set, GDB will resolve shared library path names in two stages:

First, GDB emulates ELF's dynamic section rule:

- If the shared library file path contains at least one `/` (slash), the path represents either
 - An absolute path in the target's file system, or
 - A relative path to the target's current directory (currently not supported);
- If the shared library file path contains no `/`, use the following components in order to find the file:
 1. `DT_RPATH` in the `.dynamic` section
 2. The target process's `LD_LIBRARY_PATH` environment variable
 3. The default directories `/lib/shlib` and `/usr/lib`.

If GDB cannot find the library in the preceding steps, GDB will prefix the composed file path with the `ENV_PREFIX` environment variable (if it exists). GDB uses `ENV_PREFIX` as the "virtual mount point" of the target's file system on the debug host. `ENV_PREFIX` should point to a directory on the debug host's file system which contains a duplicate image of the target's file system.

For example, if the target application uses a shared library `foo.so` with the following conditions:

- Host GDB's `_host_shlib_dirs` is
`/home/joe/proj1/usr/lib/shlib:/home/joe/ \`
`proj1/testlib`
- `DT_RPATH` is `/test/shlib:/prod/shlib`
- Target process's `LD_LIBRARY_PATH` is `/test2/shlib`
- Host's `ENV_PREFIX` is `/usr/lynx/3.1.0/mips`

GDB will look for the corresponding symbol file on the debug host in the following sequence:

1. GDB will first try `/home/joe/proj1/usr/lib/shlib/foo.so` and `/home/joe/proj1/testlib/foo.so`. If one exists and is readable, GDB will load it.
2. If the above `_host_shlib_dirs` scheme fails, GDB will try composing the following file paths using `DT_RPATH`, `LD_LIBRARY_PATH`, the default library paths, and `ENV_PREFIX`:
 - `/usr/lynx/3.1.0/mips/test/shlib/foo.so`
 - `/usr/lynx/3.1.0/mips/prod/shlib/foo.so`
 - `/usr/lynx/3.1.0/mips/test2/shlib/foo.so`
 - `/usr/lynx/3.1.0/mips/lib/shlib/foo.so`
 - `/usr/lynx/3.1.0/mips/usr/lib/foo.so`
3. If the above still fails, GDB will assume the debug host has the same directory layout as the target and will try the above file path names without `ENV_PREFIX` (`/usr/lynx/3.1.0/mips`) on the debug host.



CAUTION! Although the ELF specification allows a relative shared library path name to the application process's current working directory and LynxOS kernel-loaded shared library honors this, the current release of LynxOS GDB does not support this for automatic symbol file loading.

Application-Loaded Shared (dlopen'ed) Libraries

The LynxOS `dlopen` library resolves any application-loaded shared library path name passed to the library as an argument into a “clean” absolute path beginning with forward slash (`/`). Here, “clean” means that the path contains no single-dot (`.`)

) and double-dot (..) elements representing the current and parent directories respectively. GDB will try to locate the corresponding symbol file on the debug host's file system in the following sequence (for `/prod/shlib/foo.so`):

1. If `_host_shlib_dirs` is set, for all of its members GDB will check if a file `foo.so` exists and is readable in the directory. For example, if it is `/home/joe/proj1/usr/lib/shlib:/home/joe/proj1/testlib, /home/joe/proj1/usr/lib/shlib/foo.so` and `/home/joe/proj1/testlib/foo.so` will be checked. If one exists and is readable, GDB will load it.
2. If the above `_host_shlib_dirs` scheme fails and if `ENV_PREFIX` is set, GDB will try the given file path name prefixed by `ENV_PREFIX` (for example: `/usr/lynx/3.1.0/mips/prod/shlib/foo.so`).
3. If the above still fails, GDB will assume the debug host has the same directory layout as the target and will try the original file path name (`/prod/shlib/foo.so`) on the host's file system.

Manual Shared Library Symbol Loading/Unloading (add-symbol-file/delete-symbol-file)

For the `add-symbol-file` command, GDB accepts a file path name in the debug *host's* notation. GDB will use the exact given file path name for loading the file. For the platforms that support dynamic shared library loading, GDB will ask the target about the library's run-time loading address with the following rule:

- If the given shared library file path name is an absolute path name (starts with `/`) and it starts with the `ENV_PREFIX` directory path name, GDB will use the path name with the `ENV_PREFIX` component stripped off for the inquiry.
- If the given address is an absolute path name but does not start with the `ENV_PREFIX` directory name, GDB assumes that the host and the target have the same directory layout and uses the original absolute path name for the inquiry.
- If the given address is a relative path name (does not start with `/`), GDB will use only the file name portion (last element of the path name) for the inquiry implying "this file name in any directory."

NOTE: Path names are compared literally: GDB will not resolve traverses in the directory hierarchy with the current and parent directory notations ("`.`" and "`..`"). For example, `/usr/./abc` and `/usr/abc` will not match.

If any of the above inquiries fails, GDB considers that the shared library does not have a dynamic loading address, but is loaded statically: That is, the loading address is 0 (zero).

For each manually loaded shared library file, GDB displays a message like the following:

```
(gdb) add-symbol-file ./file.so 0
add symbol table from file "./file.so" at text_addr = 0x0
(y or n) y
Loading symbols for ./file.so with address offset 0x12345678...
```

If the target platform supports dynamic loading of shared libraries and the “address offset” is zero (0), it usually means GDB failed to determine the loading address. Check the file path for `add-symbol-file`.

Symbol Table

GDB loads and unloads symbol information by the symbol file, but it looks up the symbol table for a symbol entry by using the source file name. If two shared libraries have been built using the same source file or source files with the same name containing the same function name, and if those shared libraries are loaded at the same time, only one of the multiple functions (with the same name) is visible to GDB.

For example, if shared libraries `x.so` and `y.so` were built from `a.c` and `b.c`, and `b.c` and `c.c` respectively, a reference to function `foo` in `b.c` (`b.c:foo`) would find only one of the two functions, although there are two copies of function `foo` at different addresses. GDB currently has no way to specify the shared library file name for a symbol lookup.

Single-Stepping into a Shared Library Function

Because of the symbol scope sensitivity, the `step` command for a shared library function call may unexpectedly act like the `next` command; execution does not stop at the entry of the shared library function but it stops after returning the function call. This happens if the function belongs to a different symbol scope (another shared library) from the current one. To step into the function with stopping, set a breakpoint at the function before executing the `step` command.

Summary of Additional Commands for Shared Library Support

Refer to “Debugging with GDB” on page 11. for commands other than those listed below:

- `dbreak breakpoint_spec`
Sets a deferred breakpoint
- `ddelete [dbreakpoint_num]`
Deletes deferred breakpoints
- `info dbreak [dbreakpoint_num]`
Displays deferred breakpoints
- `add-symbol-file hostfile_name 0`
Loads an additional symbol file manually
- `delete-symbol-file hostfile_name`
Unloads a manually loaded symbol file
- `info symbol-file`
Displays symbol files currently loaded in GDB
- `info sharedlibraries`
Displays a list of shared libraries loaded to the target. This is useful only if the target process has loaded any application-loaded (dlopen'ed) shared library.

Debugging Kernel/Device Drivers

GDB can be used to debug LynxOS custom device drivers. Although GDB can be used as a tool for porting the LynxOS kernel to a new platform, it may not be very useful because LynxOS GDB kernel debugging requires a fairly stable LynxOS target kernel to operate.

Requirements

To use LynxOS GDB for kernel debug purposes, the following items are required:

- Target LynxOS with SKDB (Simple Kernel Debugger) installed
- LynxOS GDB host computer

- A serial line connection between the LynxOS target and the LynxOS GDB host or proxy server. For more details, see “Proxy Server” on page 165.

Building a Kernel for Debug Purposes

To build a LynxOS kernel for debug purposes at the source level, compile the device driver (or whatever code that will be debugged at the source level) with the `-g` option. Edit your `Makefile` to include the `-g` option for compiling (the linker does not need `-g`).

LynxWorks includes a second set of kernel libraries to help debug the kernel even more. These libraries include not only the symbols necessary for linking, but also the full debug information. By linking the kernel image with these libraries, even though the LynxOS kernel source code may not be available, one can browse some useful source level information such as calling parameters in a function call chain (stack trace).

To build a kernel image with libraries that can be debugged fully, run the `make` utility in `sys/lynx.os` with `SYS_DEBUG=true`:

```
myhost$> make SYS_DEBUG=true a.out
```

The use of the libraries with full debug information, however, increases the size of the resulting kernel image (fat image) as well as the run time memory requirement, typically by several mega bytes. If the target’s memory is tight, it is possible to strip the debug information off the fat image being loaded into the target, while still using the fat image for referencing debug information by GDB on the host.

NOTE: A stripped kernel is not capable of dynamically loading device drivers because it requires resolution of kernel symbols. To use dynamically loaded device drivers on a system where the target’s memory is tight, use the traditional kernel libraries. (Without `SYS_DEBUG=true`, you can still include full debug information for a specific device driver incurring a small to moderate increase of memory requirement.)

Debugging the Kernel

Virtually all the normal GDB features are available for kernel debugging purposes:

- Source level variable examination
- Source level singlestepping

- Call stack chain examination
- Thread support.

One big difference is that kernel debugging must always be performed in the form of remote debugging. A separate host computer to run LynxOS GDB on is required to debug the LynxOS target kernel. There is no *self* or *local* kernel debugging.

Another difference is that it is impossible to *start* and *terminate* the target kernel; GDB always interrupts a kernel that is already running to start a debug session and releases it to finish the session. This is similar to *attach* and *detach* in user process debugging. To start and finish debug sessions, see “Starting Kernel Debugging” on page 161 and “Finishing Kernel Debugging” on page 163.



CAUTION! Since kernel debugging must stop the entire operating system, the operating system will not respond when the kernel is at a breakpoint or is stopped by the debugger. In some circumstances, even though one lets the system “go,” the system may respond considerably slower or have no response at all because the debugger internally keeps single-stepping the kernel.

Simple Kernel Debugger—SKDB

LynxOS GDB on the debug host actually talks to SKDB, which is embedded in the kernel of the target with a special protocol. SKDB works as an agent and performs the following basic operations to requests made by GDB:

- Memory read
- Memory write
- Register examination
- Execution resumption
- Single-stepping.

Therefore, the target must have SKDB installed.

To build a LynxOS kernel with SKDB installed, use the `Install.skdb` utility script with the optional `SYS_DEBUG=true` flag as follows:

```
mytarget$> Install.skdb SYS_DEBUG=true
```

For more detailed information on SKDB, see Chapter 5, “Simple Kernel Debugger - SKDB”.

Threads vs. Processes

In a kernel debugging session, GDB is virtually unaware of processes at all. Though it still reports the current process ID, every thread running on the target is visible to GDB, unlike user process debugging in which only the target process's threads are visible. Therefore, a non-thread-specific breakpoint can be hit by any thread of any process on the target, including kernel threads. The `info thread` command may display a long list of threads.

Setting Up Serial Ports

Though it is possible to share the target serial port for both regular terminal use and kernel debugging, a dedicated serial port for kernel debugging is strongly recommended for reliable communication. The target serial port must have matching parameters with the host's, such as bit rate speed and parity bit. These parameters are usually configured with the target's `ttyinfo.c` file, but you can override the default values with the `stty` command on the target. To change the communication speed of GDB, use the `set remotebaud` command.

Starting Kernel Debugging

To start a kernel debug session, use the `target` command with the target of `skdb` and an appropriate protocol (serial port or `sspp` proxy).

```
(gdb) target skdb /dev/ttya
Kernel debugging using /dev/ttya
Kernel supports MTD ptrace requests.
0xdb006068 in null_loop () at main.c:224
main.c:224: No such file or directory.

(gdb)
```

NOTE: For SKDB to accept a “break-in” by GDB, the serial port must have been opened by a process. This may be done by starting a `login` process on the port by editing `/etc/ttys` or by running such a program as `cat` on the port. No user can actually log in at the port because the port is connected to the debug host. Also, the `cat` process should not output to the port. Keep the port open but quiet.

Once communication is established, GDB interrupts the target's kernel and reports the location where the kernel was interrupted, usually in the null process. In the

above example, an error message was displayed because GDB could not find the source file `main.c` in its default source file search path while the null process module was compiled with full debug information. If the source code file(s) are available, use the `dir` command to add the appropriate directory to the search path list.

If GDB returns with the following message:

```
communication not established
```

or, alternatively:

```
communication error
```

try the `target` command again. If the error persists, it may be due to wrong communication parameters or wrong target configuration such as missing SKDB or a wrong port.

Now set breakpoints at desired kernel locations and use the `continue` command to resume the kernel. Once the kernel hits a breakpoint and stops, it is possible to examine variables and the call stack chain, single-step, continue, and so forth as one would do for user process debugging.



CAUTION! Although kernel mode debugging is powerful and GDB lets the user manipulate any and all memory contents in the target system, not only those in the kernel space but also those in user process spaces, it is not advisable to set breakpoints in the user process space (user program).

The LynxOS kernel handles kernel-mode breakpoints completely different from user-mode breakpoints; a user-mode breakpoint set by kernel debugger (GDB) will not be captured by GDB running in kernel debug mode, and thus it will cause an unexpected termination of the process.

Interrupting the Kernel

To interrupt a running kernel, press **Ctrl+C** at GDB, while it is waiting for the target to stop as one would do for user process debugging. LynxOS GDB sends the break-in character to stop the target kernel.

Single-Stepping the Kernel

The `step` and `stepi` command single-step the current thread in the kernel. Unlike user process debugging, however, only the current thread can be single-

stepped; the `thread` command has no effect on single-stepping (you cannot change the current thread for singlestepping).



CAUTION! It is not recommended to attempt to trace (setting breakpoints in and/or single-stepping) the “core” portions of the LynxOS kernel, such as those handling context switching and interrupt control, because such an attempt may severely interfere with the LynxOS kernel operation.

Also, those instructions that manipulate the processor’s status register cannot be safely single-stepped. Though LynxOS GDB detects, warns about, and prevents such an attempt, casual tracing of such critical code may result in an unexpected system freeze.

Finishing Kernel Debugging

To finish a debug session and to let the target kernel resume freely, `kill` the target at the GDB prompt or quit GDB. Despite the command name, the target’s kernel is not killed, but is resumed freely.

```
(gdb) kill
Kill the program being debugged? (y or n) y
Kernel is resumed. None is actually "killed."
```

```
(gdb)
```



CAUTION! If a kernel debugging session is accidentally terminated due to a communication error or some other unexpected reason, GDB may not have had a chance to remove breakpoints before the termination. If this happens and a new kernel debug session is started, the kernel may be trapped at a breakpoint forever until the original instruction at the breakpoint location is restored by hand with a command like `print` or until the kernel is reloaded (restarted). GDB currently provides no convenient way to restore the original instructions. To reload the kernel and restart LynxOS, use the `R SKDB` command. (“Raw SKDB Commands” on page 164.) or reset the target’s hardware with the reset switch or a power-cycle.

Loading Device Drivers Dynamically

Device drivers can be dynamically loaded into memory at run-time, instead of build-time, while the system is up and running. Device drivers can be also removed

from memory and reloaded later. This facility is very convenient for device driver development.

To load a device driver dynamically, use the `drinstall` LynxOS command with the device driver's `*.o` file. To add symbol information for a dynamically loaded device driver, use the `add-symbol-file` GDB command with the driver's `*.o` file name and its loading address as reported by the `drinstall` or `drivers` LynxOS command.

```
mytarget$> drinstall -c drivers/mydriver.o
drivers/mydriver.o 0xb2000300

(gdb) add-symbol-file mytarget/drivers/mydriver.o
0xb2000300
```

Raw SKDB Commands

GDB is a generic debugger. It knows little about the LynxOS kernel. To explore the LynxOS kernel in more detail, such as the process table, thread structures, and so on, you can pass a `raw` SKDB command from LynxOS GDB using the `skdb` command followed by the desired SKDB command string.

```
(gdb) skdb p 20

pid ppid prio pgroup signals mask          sem state    name
 0  0  0  0  0  0 ffffffff  0 current  nullpr
 1  1  16  1  0  0 db168564 waiting  /init
10  1  17  10  0  0 db1228bc waiting  /bin/lpd
11  1  17  11  0  0 db17f758 waiting  /bin/login
12  1  17  12  0  0 db17b9e4 waiting  /bin/bash
13  1  17  13  0  0 db17c674 waiting  /bin/login
14  1  17  14  0  0 db17d304 waiting  /bin/login
15  1  17  15  0  0 db17df94 waiting  /bin/login
16  1  18  16  0  80000 db122abc waiting  /bin/syncer
22  1  17  22  0  0 db1232bc waiting  /net/inetd
24  1  17  24  0  0 db0e9648 waiting  /net/unfsio
27  1  17  27  0  0 db1236bc waiting  /net/portmap
29  1  17  29  0  0 db1238bc waiting  /net/mountd
31  1  17  31  0  0 db123abc waiting  /net/nfsd
33  1  17  33  0  0 db123cbc waiting  /net/rpc.statd
35  1  17  35  0  0 db123ebc waiting  /net/rpc.lockd.svc
37  1  17  37  0  0 db0e93e8 waiting  /net/rpc.lockd.clnt

tid pid prio stklen signals mask          sem state    name
 0  0  0  0  0  0 ffffffff  0 current  nullpr
 1  0  0  1096  0  0 db182314 waiting  SIMISR
 2  0  18  4168  0  0 db18d8b0 waiting  DECchip
*
```

(gdb)

See Chapter 5, “Simple Kernel Debugger—SKDB” for information on the SKDB command.



CAUTION! GDB does not keep track of any operation performed with the `skdb` command. For example, if you set a breakpoint with the `skdb` command, it would be unknown to GDB but still cause a break, and therefore the breakpoint would confuse GDB and SKDB. It is generally not advisable to use raw SKDB commands to alter the target’s state.

It is possible to change the target’s memory contents with a raw SKDB command (although it is not recommended). When doing so, it is important to clear GDB’s internal data cache with the `set remotecache` command after changing the memory contents.

Proxy Server

If GDB is set up for remote debugging over a serial line such as RS-232, the serial line usually limits the physical distance between the target and the host running GDB. LynxOS GDB extends this distance infinitely by using a proxy server—a third computer—between the target and the host. The proxy server redirects the serial line communication to a TCP/IP connection such as a local area network (LAN) or the Internet. Now you can use GDB from another room, another building, or even another country to debug the target.

The proxy server program `sspp` is a simple and small program supplied in the form of source code so that it can be ported to different platforms. The proxy server computer is either a dedicated or shared machine running a variant of UNIX including LynxOS, or it can be a terminal server that multiplexes a number of serial port connections.

The `sspp` program is transparent to the target, so it can be used for both user process debugging (`target remote`) and device driver/kernel debugging (`target skdb`).

To run the `sspp` proxy server program, the following items are required on the proxy server computer:

- A serial port connected to the LynxOS target
- TCP/IP connection to the GDB host
- BSD remote shell daemon (`rshd`)

- GDB user's account capable of using `rshd`.

These should be available on most modern UNIX workstations including LynxOS workstations.

`sspp` has been tested on LynxOS 3.1 and SunOS 4.1.x/5.x.

Syntax

The following example starts a user process debugging session (`remote`) using the `sspp` proxy server program (`sspp`) on the proxy server computer `myproxy`, whose serial port `/dev/ttya` is connected to the LynxOS target:

```
(gdb) target remote myproxy:sspp /dev/ttya
```

If the proxy program is not installed in the default search path on the server, pass its full path name to GDB. To specify the serial port communication speed, use `sspp`'s `-b` option.

The next example shows a device driver/kernel debugging session (`skdb`) using the `sspp` proxy server program loaded at `/local/bin/sspp` on the proxy server computer `myproxy`, whose serial port `/dev/com2` is connected to the LynxOS target at 19,200 bps:

```
(gdb) target skdb myproxy:/local/bin/sspp -b \
19200 /dev/com2
```

Installation

`sspp` comes in source code for easy porting to a variety of platforms. To port `sspp`, you need an ANSI-compliant C compiler (`gcc` preferred).

Compiling `sspp.c`

The associated `Makefile` is simple. Give its compile command line the desired macro definitions in the form of `-DMACRO=1`:

```
HAVE_TERMIOS
HAVE_TERMIO
HAVE_SGTTY
```

Define one and only one of these macros depending on the type of tty support of the proxy server. Both LynxOS and SunOS use `HAVE_TERMIOS`.

```
__Lynx__
```

Define this macro if the proxy server runs LynxOS.

`NO_LOCKING`

Define this macro if no locking of the serial port is required.

`LOCKF_DIR=dir`

Define this macro to override the default lock file directory
(`/var/spool/uucp`).

`MINICOM`

Define this macro if the proxy server has the Minicom terminal server installed.

Installing `sspp`

`sspp` must be installed as a set-uid executable to be able to access the serial port device file and the lock file. This requires root (super user) privilege. The `Makefile` uses the set-uid user of `uucp`. If your site has a different user ID for this purpose, change it appropriately.

After installation, try starting `sspp` from a remote machine for testing. If the results are similar to the following, the installation was successful:

```
myhost$> rsh myproxy sspp

501 Usage: sspp [-b bps] [-c host:port] [-n] /dev/ttyname
520 Terminating connection.

myhost$>
```

When modifying `sspp`'s source code for a custom environment, it can be debugged by using double colons when starting a debug session from GDB. The message transactions and some more useful information are displayed:

```
(gdb) target remote myproxy::sspp /dev/ttya
```

The proxy protocol is found in GDB's source file `ser-rsh.c`.

Minicom Terminal Server

LynxWorks uses dedicated terminal servers in its product test area. These terminal servers run the Minicom terminal server program written by Miquel van Smoorenburg on LynxOS, and multiplex serial port connections to a number of test platforms. The Minicom terminal server program is usually installed as the user's login shell and thus it is impossible for the user to use remote shell for the `sspp`

proxy. The `sspp.arb` script arbitrates between Minicom and `sspp` by checking if a login is interactive (Minicom) or remote (`sspp`) with a simple time-out mechanism.

To use this arbitrator script, first edit the script so that the function `gominicom` points to a correct path for Minicom. Then install it as the user's login shell (put the script's path into the login shell field of the user's `passwd` entry).

General Tips and Miscellaneous Issues

The following sections provide you with general tips and other useful information:

- Reading and writing large memory blocks
- Executing remote shell commands
- Browsing target process's environment
- Function calls in a multithreaded process
- Function calls after **Ctrl+C**
- Resuming off a blocking system call
- Debugging a signal-intensive process.

Reading and Writing Large Memory Blocks

The `memget` and `mempout` commands let you transfer large blocks of memory contents from and to, respectively, the target process's address space very efficiently. Table 3-4 shows the syntax for `memget` and `mempout` commands.

Table 3-4: Reading and Writing Large Memory Blocks

Syntax	Description
<code>memget localfile address size</code>	<i>localfile</i> is the GDB host's local file to which the memory block is transferred. <i>address</i> is the absolute address in the debugged process's address space in either decimal, hexadecimal, or octal (no symbolic address is allowed). <i>size</i> is the transferred memory size in number of bytes in decimal.
<code>memget ">localfile" address size</code>	Same as the first syntax except that the surrounding quotation marks are mandatory.
<code>memget "/localcommand args" address size</code>	Same as the first syntax except that it redirects the memory bytes to <i>localcommand</i> 's standard input through pipe. The surrounding quotation marks are mandatory.
<code>memput localfile address size</code>	Same as the first syntax except that <i>localfile</i> is the GDB host's local file from which the memory block is transferred. <i>size</i> is optional and the default is <i>localfile</i> 's size.
<code>memput "<localfile" address size</code>	Same as the previous syntax. <i>size</i> is optional and the default is <i>localfile</i> 's size. The surrounding quotation marks are mandatory.

Browsing Target Process's Environment

The `info environment` command displays the debug target process's environment.

```
(gdb) info environment PATH
/usr/local/bin:/usr/bin:/bin:.
```

If the debug target process is not available the `info environment` command displays the debugger's or `gdbserver`'s environment, depending on the mode of debugging (local or remote).

Executing Remote Shell Commands

The `rshell` command lets one execute a shell command on the remote target machine. It is similar to the BSD `rsh` (remote shell) command, but `rshell` works

even over a serial connection without TCP/IP. Therefore, it is useful for browsing the embedded target's directories, processes, and so forth.

```
(gdb) rshell ls -lF
total 80
-rw-r--r--  1 joe      34268   Jan 2  10:45  core
-rw-r--r--  1 joe       152   Oct 20  1999  getarc
drwxr-xr-x  8 joe        512   Nov 8  1997  src/
```

The `rshell` command handles quotation marks and other shell meta characters properly if the target has a shell program installed; otherwise, it does not.



CAUTION! Do not execute an interactive program or a program that reads the standard input with the `rshell` command. The standard input of the remote program is redirected to `/dev/null`.

In remote debugging, the `rshell` command becomes effective only after a connection with the target is established with the `target` command. If the `rshell` command is used before a connection is established or after connection is lost, it works for the local host because the default target is the local host.

Function Calls in a Multithreaded Process

To manually execute a function call, such as a command line like “`print foo(1)`” at the GDB prompt, GDB performs the following:

1. Allocates a block of memory in the current thread's stack in the target process (moves the thread's stack pointer value)
2. Writes a piece of “caller” code that makes a call to the target function (“`foo`” in the above example) with necessary argument handling and ends with a breakpoint instruction in the target's extended stack area
3. Substitutes the current thread's program counter value temporarily to the caller code's entry address
4. Resumes the target process (including the current thread) and waits for the target process to stop (hopefully by hitting the breakpoint instruction in the caller code)
5. After control returns to GDB, restores all the register values of the thread (the extended stack area is discarded).

It should be noted that if a function is called by hand in a target process that has more than one thread, not only the current thread but also all other sibling threads

in the target process are subject to scheduling, as described in “Resuming Threads” on page 140. This usage may result in unexpected side effects if some other thread actually runs while the manual function call is running. To prevent this, raise the current thread’s priority temporarily as described in “Resuming Threads” on page 140 provided that the function call does not involve a blocking system call.

Functions Calls after Ctrl+C

If the current thread is in a blocking system call and you type **Ctrl+C** from GDB to interrupt the thread, the target process stops and control returns to GDB. If you then try to call a function in the target program by hand with the `print` command or a similar command, GDB appears to be hung and the function will not be executed.

This is because the target process has to complete the blocking system call before running the function. The hung function may be interrupted by entering another **Ctrl+C**. Then, the `continue` command resumes the target process to reenter the system call.

Resuming after a Blocking System Call

If the current thread stops at a system call instruction due to a breakpoint and you try to resume the application process with the `continue` command, GDB may report the following warning:

```
SIGNONBLOCK, Would deadly block
```

In this case, GDB regains control without executing the blocking system call. To resume the process continuously, remove the cause of the blocking.

This is a result of GDB’s internal mechanism to resume a thread from a breakpoint. When GDB resumes a process that has stopped due to a breakpoint hit by one of its threads, GDB:

1. Restores the original instruction at the breakpoint location
2. Lets the thread single-step the instruction
3. Reinstalls the breakpoint instruction there
4. Resumes the thread freely, if needed.

In order to ensure an atomic operation, LynxOS GDB uses special single-stepping which guarantees that only the thread in the process runs while all other sibling threads in the same target process remain stopped. Otherwise, some other thread in

the same process may run while the original instruction at the breakpoint location is restored and that take-over thread may miss the breakpoint.

In this scenario, if the breakpoint were set at a system call instruction that waits for a resource that is locked by another thread in the same target process (for example, mutex) the system call would never complete because no other threads in the target process can run, and thus this would cause a dead lock. The LynxOS kernel prevents such a dead lock by detecting the situation and breaking the blocking with a special internal signal (`SIGNONBLOCK`).

Debugging a Signal-Intensive Process

LynxOS relies on the UNIX/POSIX 1 synchronous signal mechanism for debugging a process: Any signal to a traced (debugged) process is captured by the kernel's process trace code, causing the traced process to stop. The stop is then reported to the debugger. A breakpoint or single-stepping merely generates and sends a special signal to the traced process.

Although GDB can be configured with the `handle` command to pass and not print a signal when it is sent to the target (traced) process, GDB implements this signal handling by software: Any signal sent the target process still causes the process to stop; when GDB detects the target process's stop, it examines the signal code that caused the stop; If the signal code is configured to pass, GDB silently resumes the target process without reporting it to the user.

Obviously, the above processing involves software overhead in both the kernel and GDB. If the target process is designed to receive signals frequently, its execution speed may noticeably slow down under a debugger, even though the signals are not explicitly reported by GDB. Remote debugging makes the situation even worse because of the additional overhead for the remote communication transaction for each signal reception.

CHAPTER 4 *Debugging with Total/db*

Total/db is the LynuxWorks debugger which is based on the GNU GDB debugger and Insight graphical user interface. LynuxWorks has added many customizations and enhancements to the standard GDB debugger so that it may be used more efficiently with the LynxOS operating system. Total/db is capable of remote debugging of LynxOS kernels and applications as well as multithreaded debugging.

This chapter discusses the Insight user interface. Cygnus Insight is a graphical user interface for GDB, the GNUPro Debugger. Insight has the same look and feel on both Windows and Unix operating systems. Insight offers the ease of a GUI and access to all the power of the GDB's command-line interface.

Source Window

When Insight first opens, it displays the **Source Window** (see Figure 4-1).

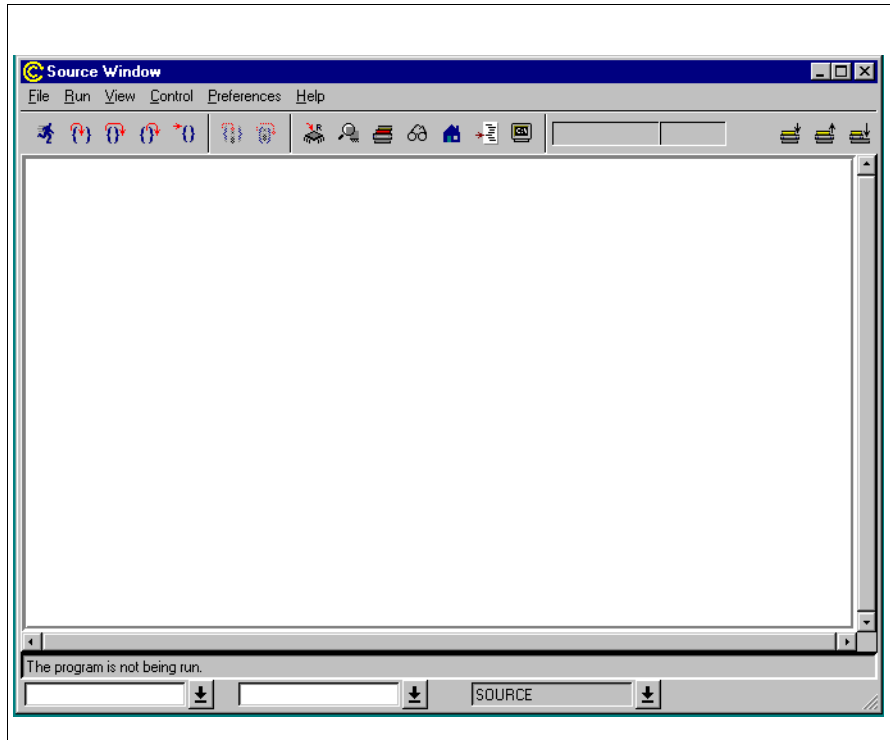


Figure 4-1: Source Window

The **Source Window** menu bar has the following items: **File**, **Run**, **View**, **Control**, **Preferences** and **Help**.

File Menu

Figure 4-2 shows the **File Menu**.

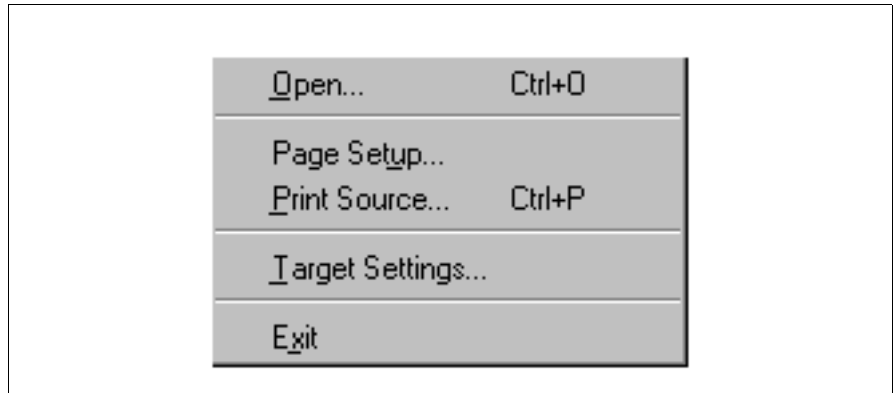


Figure 4-2: File Menu

The **File Menu** options are:

Open	Brings up the Load New Executable dialog box. See “Load New Executable Dialog Box” on page 189.
Page Setup	Brings up the Page Setup dialog box. See “Page Setup Dialog Box” on page 191. (This option is currently not available on the Unix version.)
Print Source	Brings up the Print dialog box. See “Print Dialog Box” on page 192. (This option is currently not available on the Unix version.)
Target Settings	Brings up the Target Settings dialog box. See “Target Selection Dialog Box” on page 192.
Exit	Closes the Insight program.

Run Menu

Figure 4-3 shows the **Run Menu**.



Figure 4-3: Run Menu

The **Run Menu** options are:

- Download** Downloads a program to a board (if connected).
- Run** Runs the executable program.

View Menu

Figure 4-4 shows the **View Menu**.

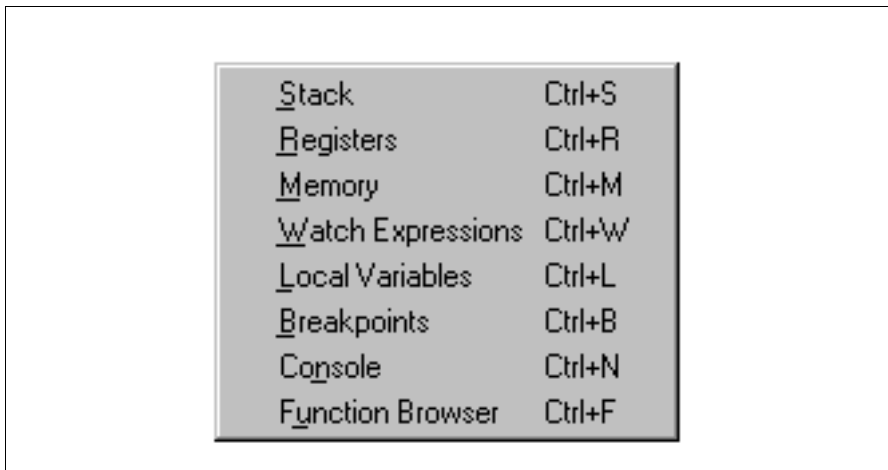


Figure 4-4: View Menu

The **View Menu** options are:

Stack	Displays Stack window. See “Stack Window” on page 197.
Registers	Displays Registers window. See “Registers Window” on page 198.
Memory	Displays Memory window. See “Memory Window” on page 200.
Watch Expressions	Displays Watch Expressions window. See “Watch Expressions Window” on page 202.
Local Variables	Displays Local Variables window. See “Local Variables Window” on page 205.
Breakpoints	Displays Breakpoints window. See “Breakpoints Window” on page 206.
Console	Displays Console window. See “Console Window” on page 209.
Function Browser	Opens the Function Browser window. See “The Function Browser Window” on page 210.

Control Menu

Figure 4-5 shows the **Control Menu**.

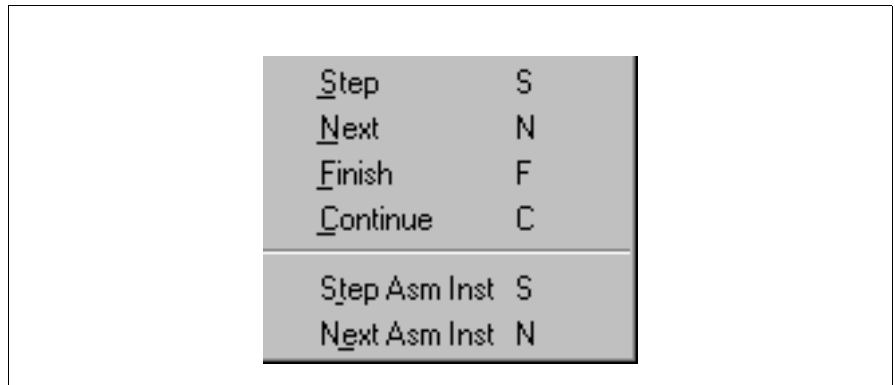


Figure 4-5: Control Menu

The **Control Menu** options are:

Step	Steps to next executable line of source code. Steps into called functions.
Next	Steps to next executable line of source code in current file. Steps over called functions.
Finish	Finishes execution of the current frame. If clicked while in a function, it finishes the function and returns to the line that called the function.
Continue	Continues execution until a breakpoint, watchpoint or other exception is encountered; or execution is complete.
Step Asm Inst	Steps to next assembler instruction. Steps into subroutines.
Next Asm Inst	Steps to next assembler instruction. Executes subroutines and steps to the subsequent instruction.

Preferences Menu

Figure 4-6 shows the **Preferences Menu**.



Figure 4-6: Preferences Menu

The **Preferences Menu** options are:

Global	Displays Global Preferences dialog box. See “Global Preferences Dialog Box” on page 195.
Source	Displays Source Preferences dialog box. See “Source Preferences Dialog Box” on page 196.

Help Menu

Figure 4-7 shows the **Help Menu**.

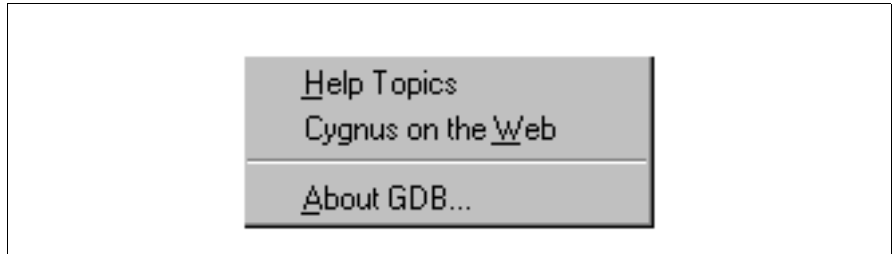


Figure 4-7: Help Menu

The **Help Menu** options are:

Help Topics	Displays Help window.
Cygnus on the Web	Links to the GNUPro Tools web page.
About GDB...	Displays About GDBTk window, containing product version number, copyright and Cygnus contact information for Insight.

Toolbar Buttons

The toolbar provides quick access to various debugger functions. Table 4-1 list the toolbar buttons.

Table 4-1: Toolbar Buttons








Icon	Name	Description
	Run	Runs the executable. During execution the button turns into the Stop button. If you click on the Run button with no executable loaded, you invoke the Target Selection dialog box. See “Target Selection Dialog Box” on page 192.
	Stop	Interrupts the program, provided that the underlying hardware and protocol support this functionality. Many monitors that are connected to boards cannot interrupt programs on those boards. In this case, the Stop button has no functionality.
	Step	Steps to next executable line of source code. Steps into called functions.
	Next	Steps to next executable line of source code in the current file. Steps over called functions.
	Finish	Finishes execution of the current frame. If clicked while in a function, it finishes the function and returns to the line that called the function.
	Continue	Continues execution until a breakpoint, watchpoint or other exception is encountered; or execution is complete.
	Step Assembler Instruction	Invokes step assembler instruction. Steps into subroutines.

Table 4-1: Toolbar Buttons (Continued)









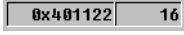



Icon	Name	Description
	Next Assembler Instruction	Steps to next assembler instruction. Executes subroutines and steps to the following instruction.
	Register	The Registers button brings up the Registers window. See “Registers Window” on page 198
	Memory	The Memory button brings up the Memory window. See “Memory Window” on page 200.
	Stack	The Stack button brings up the Stack window. See “Stack Window” on page 197.
	Watch Expressions	The Watch Expressions button brings up the Watch Expressions window. See “Watch Expressions Window” on page 202.
	Local Variables	The Local Variables button brings up the Local Variables window. See “Local Variables Window” on page 205.
	Breakpoints	The Breakpoints button brings up the Breakpoints window. See “Breakpoints Window” on page 206.
	Console	The Console button brings up the Console window. See “Console Window” on page 209.

Table 4-1: Toolbar Buttons (Continued)

Icon	Name	Description
	Line Address & Line Number Display	The left side displays the program counter of the current frame, while the program is running. The right side displays the line number, which contains the program counter, while the program is running.
	Down Stackframe	Moves down the stack frame one level.
	Up Stackframe	Moves up the stack frame one level.
	Go to Bottom of Stack	Moves to the bottom of the stack frame.

Special Display Pane Features

- When the executable is running, the location of the current program counter is displayed as a line with a green background.
- When the executable has finished running, the background color changes to violet (browsing mode).
- When looking at a stack backtrace, the background color changes to golden yellow.

Using the Mouse in the Display Pane

There are various uses of the mouse within the main display pane of the **Source Window**. The display pane is divided into two columns (see Figure 4-8). The left column extends from the left edge of the display pane to the last character of the line number. 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 a different set of effects.

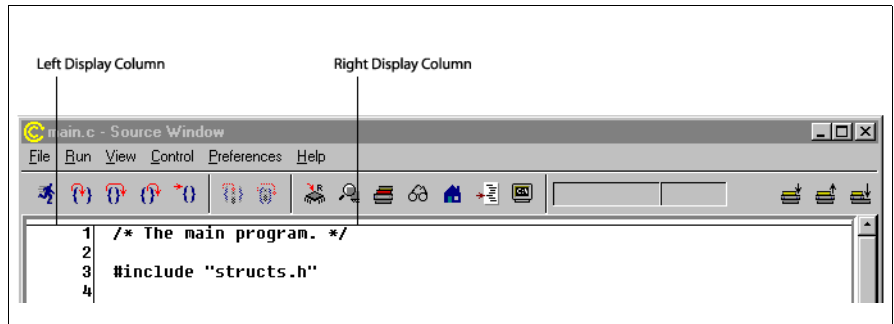


Figure 4-8: Using the Mouse in the Window

Right Display Column

- By holding the cursor over a global or local variable, the current value of that variable is displayed.
- By holding the cursor over a pointer to a structure or class, the type of structure or class is displayed and the address of the structure or class is displayed.
- By double clicking an expression, it is selected.
- By right clicking while an expression is selected, a pop-up menu appears (see Figure 4-9). The selected expression appears in both menu selections.



Figure 4-9: Pop-up Window for Expressions

The pop-up menu options are:

The Add <i>var</i> to Watch	Brings up the Watch Expressions Window and adds a variable expression (the <code>lis</code> variable, in this instance) to the list of expressions in the window. See “Watch Expressions Window” on page 202.
Dump Memory at <i>var</i>	Brings up the Memory Window , which displays a memory dump at an expression, in this instance, the <code>lis</code> expression. See “Memory Window” on page 200.

Left Display Column

When the cursor is in the left column and it is over an executable line (marked on the far left by a minus sign), it changes into a circle. When the cursor is in this state, events have the following results:

- A left click sets a breakpoint at the current line. The breakpoint appears as a red square in place of the minus sign.
- A left click on any existing breakpoint or temporary breakpoint removes that breakpoint.
- A right click brings up another pop-up menu (see Figure 4-10) for setting breakpoints.

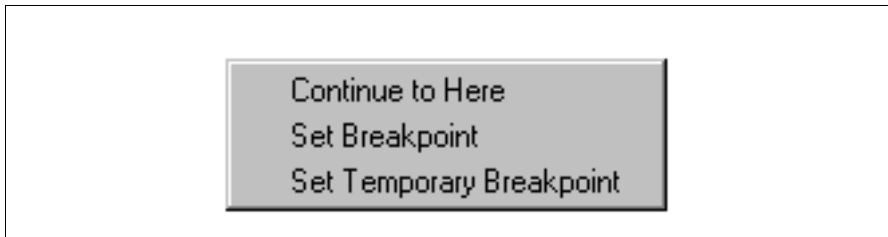


Figure 4-10: Pop-up Menu for Setting Breakpoints

The pop-up menu options are:

- | | |
|-------------------------|--|
| Continue to Here | This causes the program to run up to this location, ignoring any breakpoints. Like the temporary breakpoint, this menu selection is displayed as an orange square. This selection disables all other breakpoints. When a breakpoint has been disabled, it turns from red or orange to black. |
|-------------------------|--|

NOTE: The debugger might be expected to execute to a given location, stopping at all encountered breakpoints. This menu item currently forces execution to this location without stopping at any encountered breakpoints

Set Breakpoint	This sets a breakpoint on the current executable line. This has the same action as left clicking on the minus sign.
Set Temporary Breakpoint	This sets a temporary breakpoint on the current executable line. A temporary breakpoint is displayed as an orange square. The temporary breakpoint is automatically removed when it is hit.

Figure 4-11 shows the pop-up menu for deleting breakpoints.

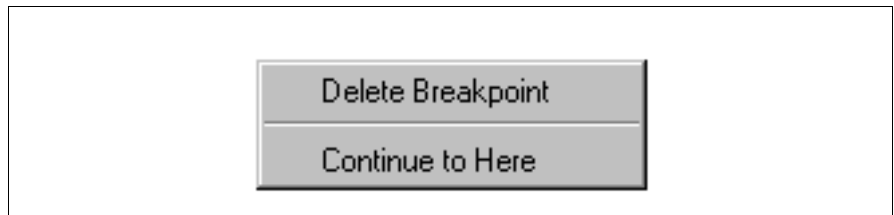


Figure 4-11: Pop-up Menu for Deleting Breakpoints

The menu options are:

Delete Breakpoint	This deletes the breakpoint on the current executable line. This has the same action as left clicking on the red square.
Continue to Here	This causes the program to run up to this location, ignoring any breakpoints. Like the temporary breakpoint, this menu selection is displayed as an orange square. This selection disables all other breakpoints. When a breakpoint has been disabled, it turns from red or orange to black.

Below the Horizontal Scroll bar

There are four display and selection fields below the horizontal scroll bar: the status text box, the drop-down list box, the function drop-down combo box and the code display drop-down list box.

Status Text Box

At the top of horizontal scroll bar, a text box displays the current status of the debugger (in the status box for the window depicted in Figure 4-12: Status text box, the message reads "Program stopped at line 19" as current status for the example program.)

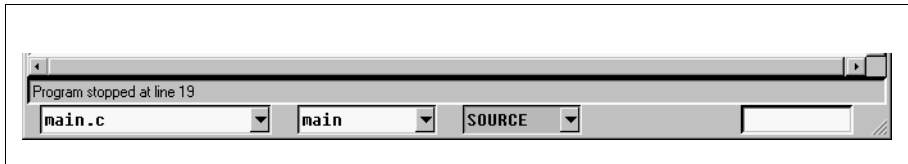


Figure 4-12: Status Text Box

Function List and Combo Boxes

Figure 4-13 shows the drop-down list box. The drop-down list box displays all the source (.c) and header (.h) files associated with the executable. Files may be selected by clicking in the list box, or by typing into the text field above the list. The drop-down list box displays all the functions in the currently selected source or header file. A function may be selected by clicking in the list box, or by typing into the text field above.

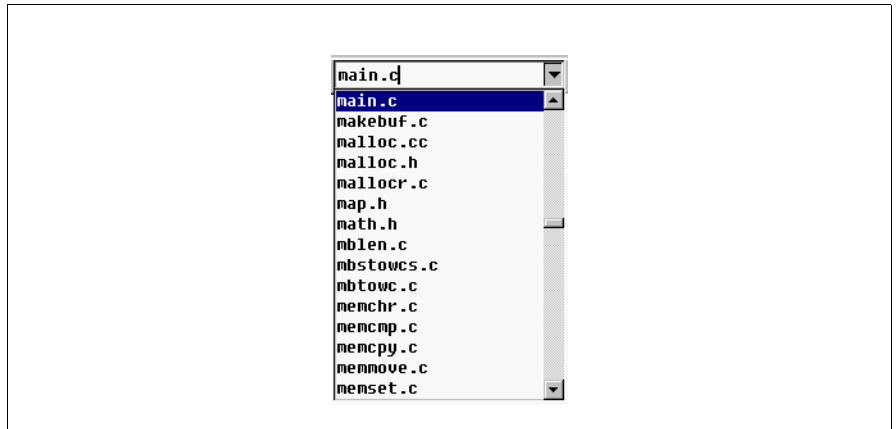


Figure 4-13: Drop-down List Box

For the function drop-down combo box, the `main.c` file only contains the one 'main' function. Figure 4-14 function drop-down combo box.

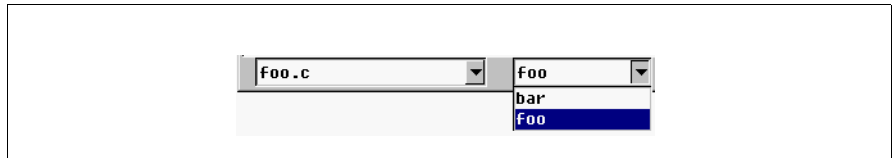


Figure 4-14: Function Drop-down Combo Box

Code Display List Box

Figure 4-15 shows the code display drop-down list box.

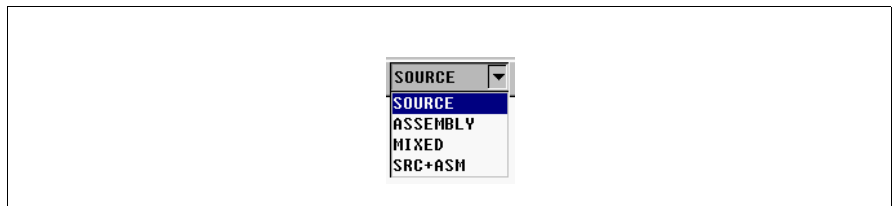


Figure 4-15: Code Display Drop-down List Box

Use the code display drop-down list to select how the code in the **Source Window** is displayed. The options are:

SOURCE	The source code is displayed in the Source Window .
ASSEMBLY	The assembly code is displayed in the Source Window .
MIXED	The source code and the assembly code are both displayed, interspersed in the Source Window .
SRC+ASM	The source code and the assembly code are both displayed in a double paned window. The source code is displayed in the Source Window and, in a pane below the source code pane, the assembly code is displayed.

Search Text Box

Figure 4-16 shows the search text box. By typing into the search text box and pressing **Enter**, a forward search is done on the source file for the first instance of the character string entered. By pressing the **Shift** and **Enter** keys simultaneously, a backward search is performed. Repeatedly hitting **Enter** or the **Shift** and **Enter** keys simultaneously, repeats the search forward or backward in the search window.



Figure 4-16: Search Text Box

If you type "@" in the search text box and a number, the source display jumps to the line of the number specified. For instance, after having specified "@" and "6" in the search text box, the example program shows a jump to line 6 in the search text dialog box (see Figure 4-17).

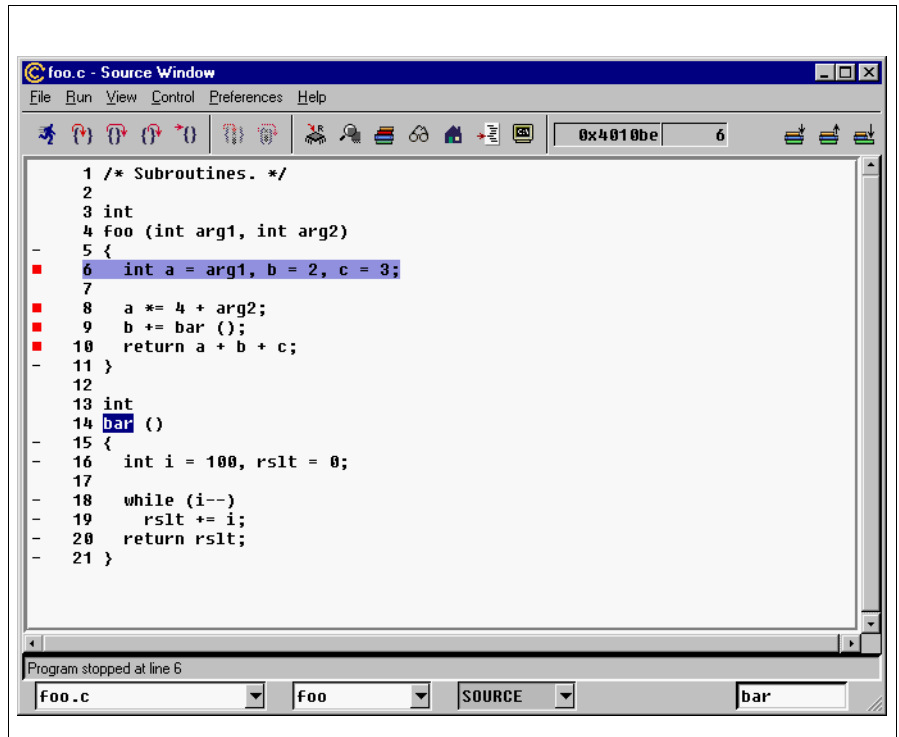


Figure 4-17: Using the Search Text Dialog Box

Dialog boxes for the Source Window

The section describes the **Source Window** dialog boxes.

Load New Executable Dialog Box

The **Load New Executable** dialog box (see Figure 4-18) is invoked by clicking **Open** from in the **File Menu**. This dialog box allows you to navigate through directories and select an executable file to be opened in the **Source Window**.

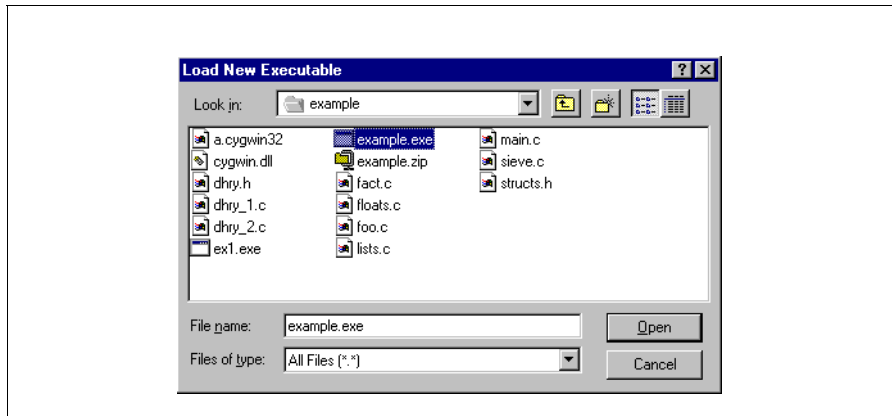


Figure 4-18: Load New Executable Dialog Box

Page Setup Dialog Box

The **Page Setup** dialog box (see Figure 4-19) is invoked by clicking **Page Setup** from the **File Menu**. This dialog box allows you to make page layout selections before printing a source file.

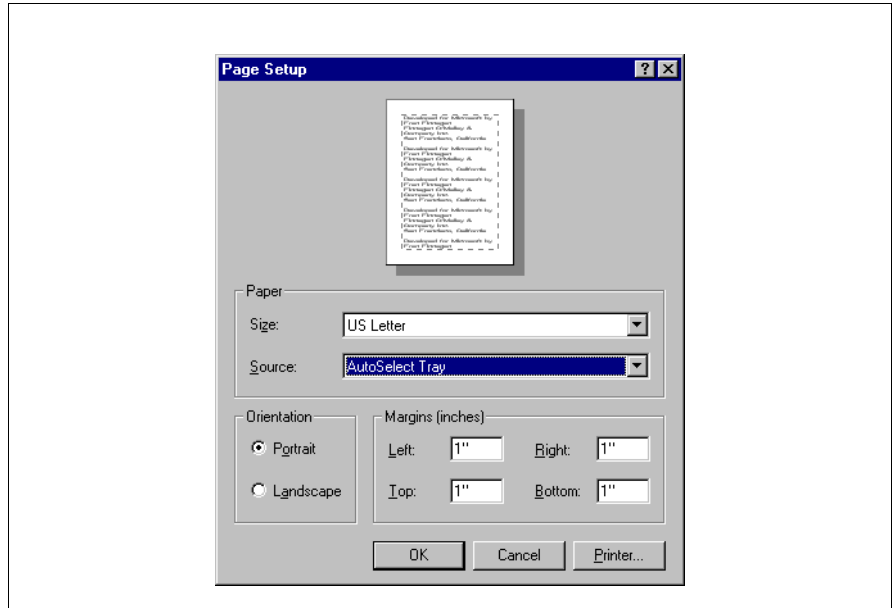


Figure 4-19: Page Setup Dialog Box

Print Dialog Box

The **Print** dialog box (see Figure 4-20) is invoked by clicking **Print Source** from the **File Menu**. This dialog box allows you to select a printer and make other print specific selections, before printing a source file.

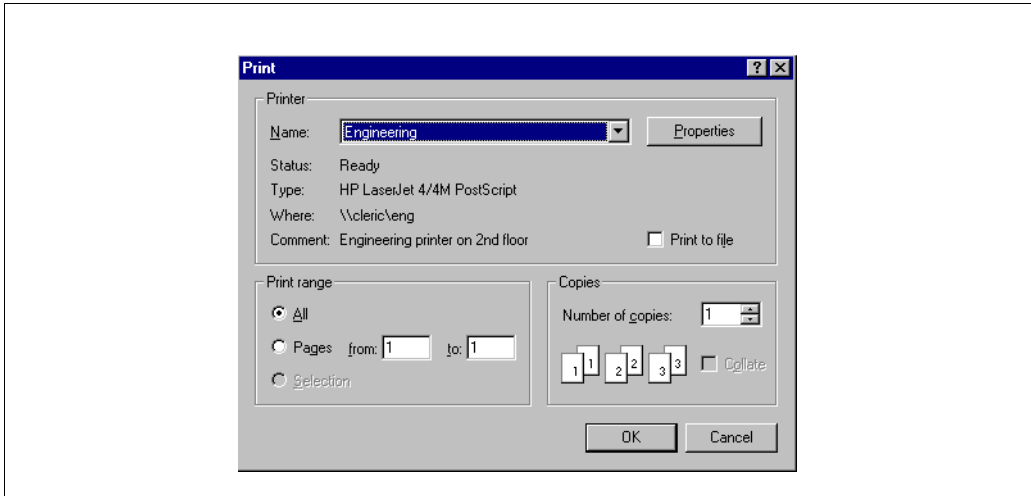


Figure 4-20: Print Dialog Box

Target Selection Dialog Box

The **Target Selection** dialog box (see Figure 4-21) is invoked by clicking **Target Settings** from the **File Menu**. This dialog box allows you to select the target you wish to run the executable on, and make other run specific selections.

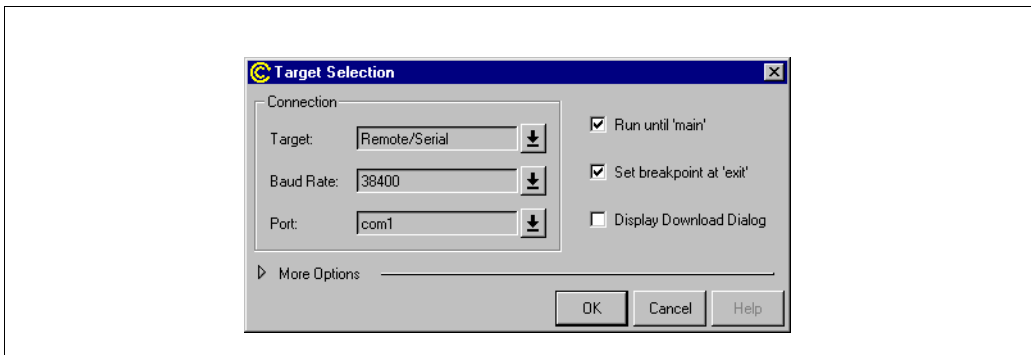


Figure 4-21: Target Selection Dialog Box

The basic set of options include:

Connection	The Connection group contains the target drop-down list box for target selection and two other fields for setting target-specific parameters.
Target	The contents of this list box depends upon the specific GDB debugger configuration you have received. For a native configuration, the list contains Exec (for native execution), Remote/Serial (serial connection to a remote target) and Remote/TCP (TCP connection to a remote target). If GDB has been configured to include a specific hardware simulator, the target Exec will be replaced by target sim . The names of specific hardware targets may also be included in the list, with serial, TCP or both methods of connection, depending upon the hardware.
Baud Rate/ Hostname	When a serial connection to a remote target is selected the baud rate may be set. When a TCP connection to a remote target is selected, this list box turns into a text edit field, renamed Hostname , allowing for specifying of a host name.
Port	For both serial and TCP connections to remote targets, the port must be designated. For serial connections, port specifies the serial port on the host machine. For TCP connections, port specifies the port number on the remote target.
Run until 'main'	Set a breakpoint at main and run until that breakpoint is reached. This is checked by default.
Set breakpoint at 'exit'	Set a breakpoint at the call to the 'exit' routine. This is checked by default.
Display Download Dialog	In addition to using the status-bar, display more extensive download status information in a dialog box. This is particularly useful when doing a serial download to a remote target. This is unchecked by default.

More Options /Fewer Options

The **More Options/Fewer Options** selection of the **Target Selection** dialog box toggles to display or hide the **Run Options** at the bottom of the dialog box (see Figure 4-22).

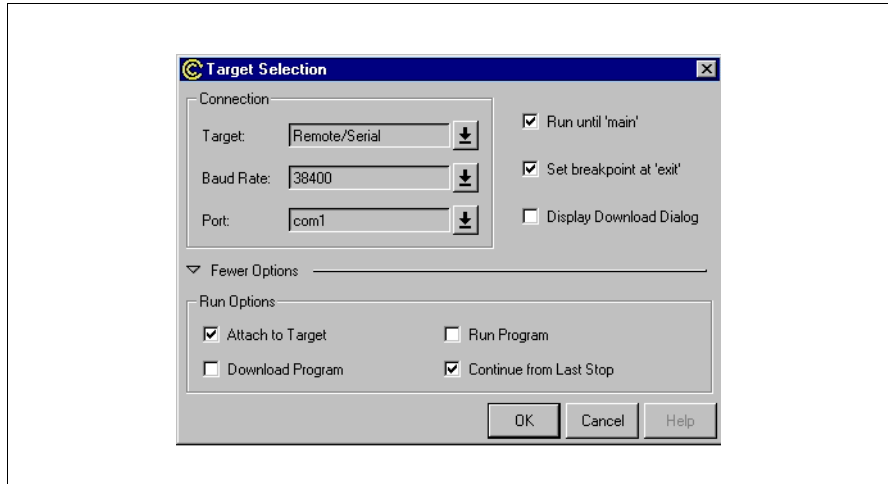


Figure 4-22: Run Options

The four check boxes in the **Run Options** group set-up the actions taken, when the **Run** button is clicked. The run options include:

- | | |
|--------------------------------|---|
| Attach to Target | Connects to a remote target. |
| Download Program | Downloads an executable to a remote target. |
| Run Program | Begins execution of an executable. |
| Continue from Last Stop | Continues execution from wherever the executable, on a remote target, left off. |

The **Fonts** group allows for custom selection of font family and size. The options include:

- Fixed Font** This drop-down list box allows you to select the font for the source code display panes.
- Default Font** This drop-down list box allows you to select the font for use in list boxes, buttons and other controls.
- Statusbar Font** This drop-down list box allows you to select the font for the status bar.

Source Preferences Dialog Box

The Source Preferences dialog box (see Figure 4-26) is invoked by clicking **Source** from the **Preferences Menu**.

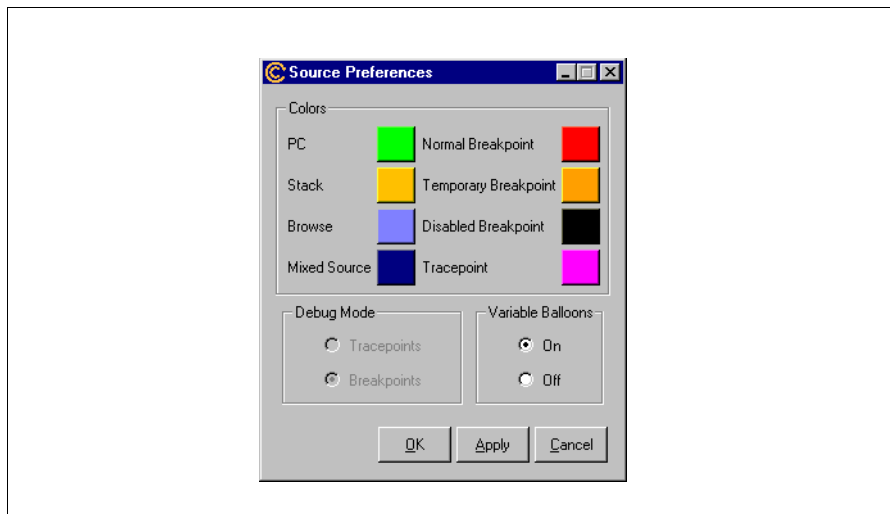


Figure 4-26: Source Preferences Dialog Box

The source preferences options are:

- Colors** Single left-clicking any of the colored squares opens the **Choose** color dialog box. The **Choose** color dialog box allows the display colors to be modified by the user.

Debug Mode	Unless GDB has been configured to enable the setting of trace points, this radio button has no effect.
Variable Balloons	If Variable Balloons is on, a balloon appears displaying the value of a variable when the mouse is placed over the variable in the Source Window . The default setting is On .

Stack Window

The **Stack** window (see Figure 4-27) displays the current state of the call stack. Each line represents a stack frame.

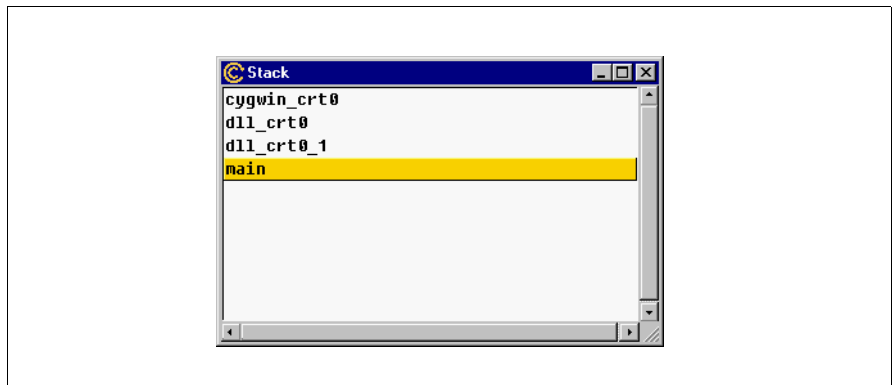


Figure 4-27: Stack Window

Clicking a frame selects that frame, indicated by the background of the frame turning yellow. The **Source Window** automatically updates to display the line, corresponding to the selected frame. If the frame points to an assembly instruction, the **Source Window** changes to display assembly code. The background of the corresponding line in the **Source Window** also changes to yellow.

Registers Window

The Registers window (see Figure 4-28) dynamically displays the registers and their content.

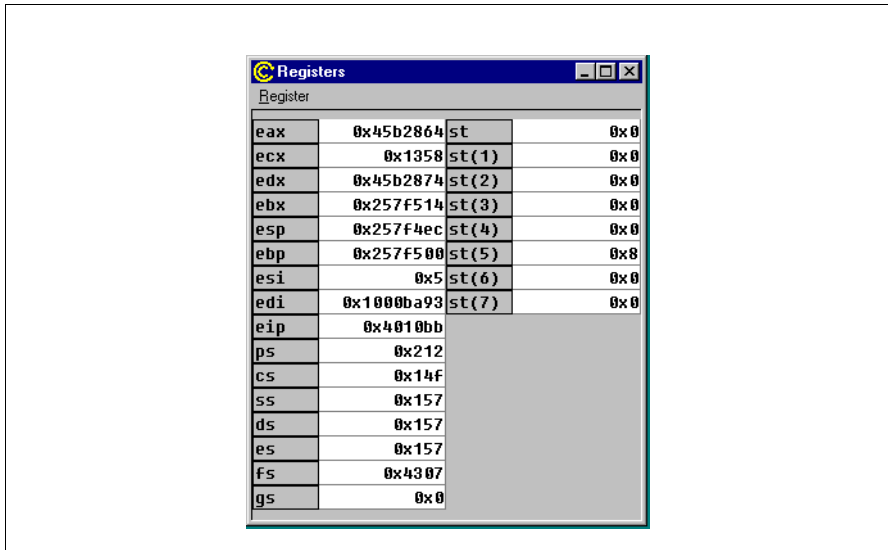


Figure 4-28: Registers Window

A double click on a register allows the content of the register to be edited. Hitting the escape key (Esc) will abort the editing.

Changing register properties is handled by way of the **Register** menu (see Figure 4-29).

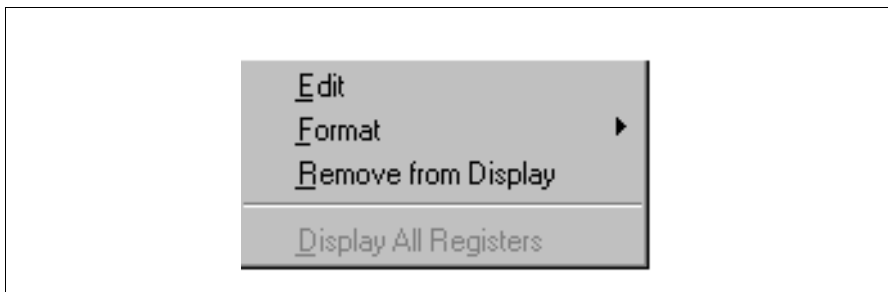
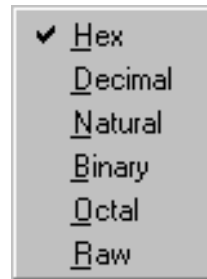


Figure 4-29: Register Menu

The **Register** menu options are:

- Edit** This menu item has the same effect as double clicking a register. The content of the selected register may be changed. This menu item is only active when a register has been selected.
- Format** This menu item calls another pop-up menu, as shown below, allowing the content of the selected register to be displayed in hexadecimal, decimal, natural, binary, octal, and raw formats. **Hexadecimal (Hex)** is the default display format.



- Remove from Display** This menu item removes the selected register from the window. All registers are displayed if the window is closed and reopened. This menu item is only active when a register has been selected.
- Display All Registers** This menu item displays all the registers. This menu item is only active when one or more registers have been removed from display.

Memory Window

The Memory window (see Figure 4-30) dynamically displays the state of memory.

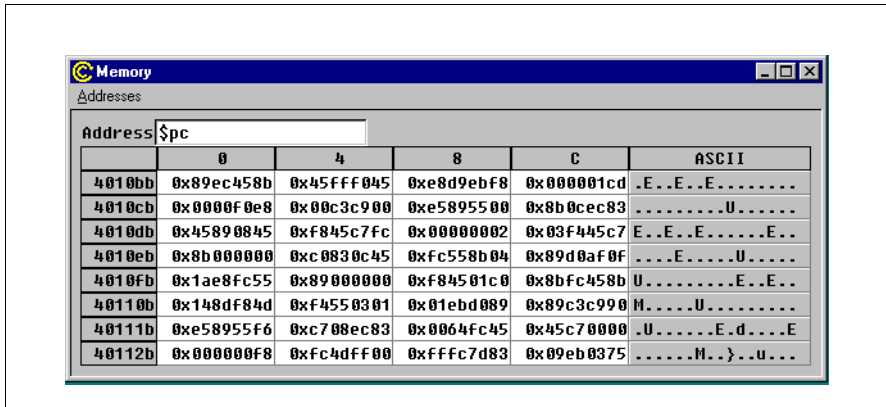


Figure 4-30: Memory Window

A memory location can be selected by double clicking the left mouse button with the cursor in the window. The contents of a selected memory location can be edited.

The Addresses menu is shown in Figure 4-31.

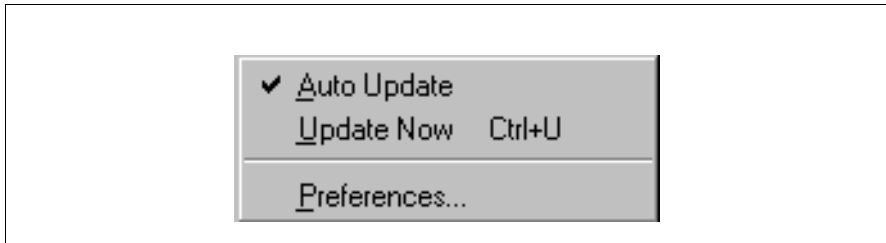


Figure 4-31: Address Menu

The **Addresses** menu options are:

- Auto Update** The contents of the **Memory** window are automatically updated whenever the state of target changes. This is the default setting.
- Update Now** Forces the immediate update of the **Memory** window's view of the target's memory.
- Preferences** This menu item brings up the **Memory Preferences** dialog box.

Memory Preferences Dialog Box

The **Memory Preferences** dialog box (see Figure 4-32) makes it possible to set memory options.

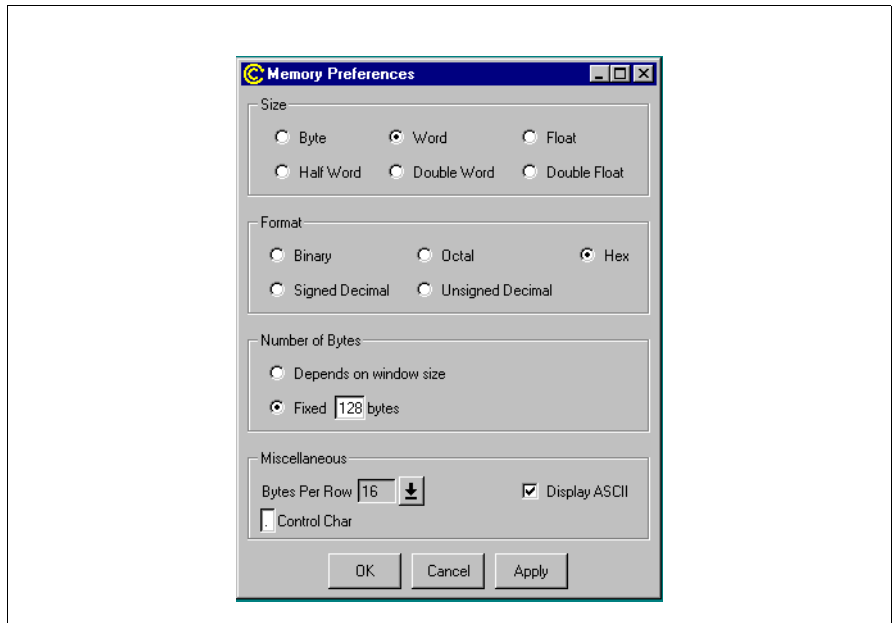


Figure 4-32: Memory Preferences Dialog Box

The memory preference options are:

- Size** Selection of the size of the individual cells displayed.
- Format** Selection of the format of the memory display.

- Number of Bytes** Sets the number of bytes displayed in the **Memory** window.
- Bytes Per Row** Sets the number of bytes displayed per row.
- Display ASCII** Choose to display a string representation of the memory.
- Control Char** Choose the character used to display non-ASCII characters. The default character is the period.

Watch Expressions Window

The **Watch Expressions** window (see Figure 4-33) displays the name and current value of user-specified expressions.

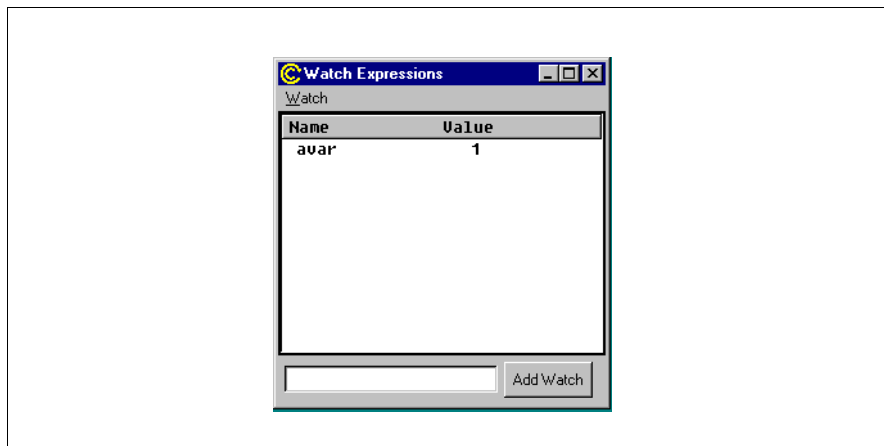


Figure 4-33: Watch Expressions Window

- Single clicking on an expression selects that expression.

- Right clicking in the display pane, while an expression is selected, calls an expression specific **Watch** menu, as shown in Figure 4-34.

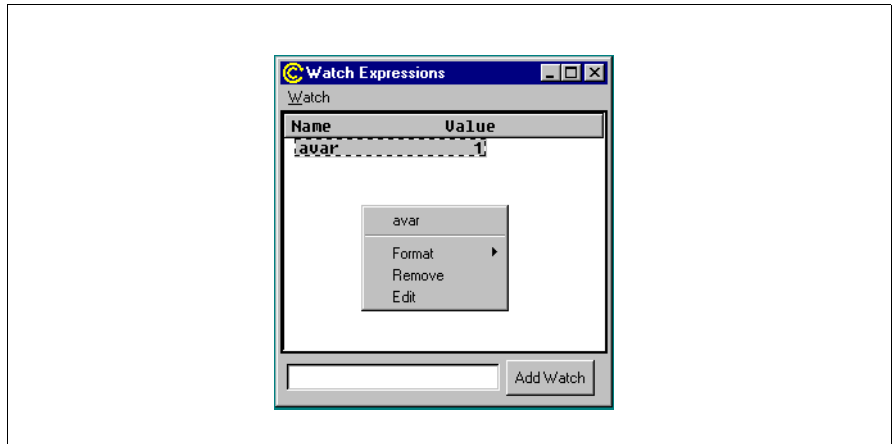
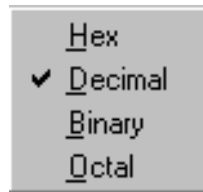


Figure 4-34: Expression Specific Watch Menu

The **Watch Expressions** menu options are:

- Edit** Allows the value in the expression to be edited. Hitting the escape key (Esc) will abort the editing.
- Format** This menu item brings up another pop-up menu, as shown below, allowing the value of the selected expression to be displayed in hexadecimal, decimal, binary, or octal formats. By default, pointers are displayed in hexadecimal and all other expressions are displayed as decimal.



- Remove** Removes the selected expression from the watch list.

Add Watch Button

An expression can be typed into the text edit field at the bottom of the dialog box, as shown in the left screen of Figure 4-35. By pressing the **Add Watch** button or hitting the **Enter** key, the expression is added to the list, as shown in the right screen of Figure 4-35. Invalid expressions are ignored.

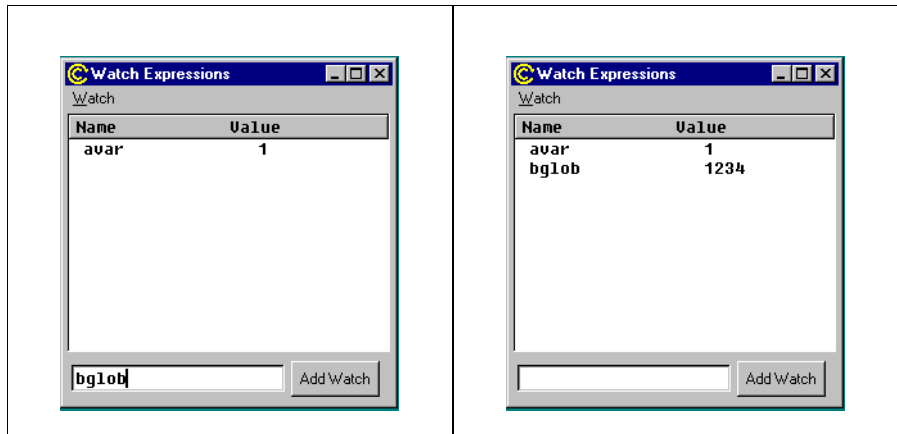


Figure 4-35: Add Watch Button

Watching Registers

GDB allows registers to be added to the **Watch Expressions** window, by typing register convenience variables into the text edit field. Every register has a corresponding convenience variable. The register convenience variables consist of a dollar sign followed by the register name. The convenience variable for the program counter is `$pc`, for example. The convenience variable for the frame pointer is `$fp`.

Casting Pointers in the Watch Expressions Window

Pointer values may be cast to other types and watched, represented as the type to which the pointer was cast. For example, by typing `(struct _foo *) bar` in the text edit field, the `bar` pointer is cast as a `struct _foo` pointer.

Local Variables Window

The **Local Variables** window displays the current value of all local variables (see Figure 4-36).

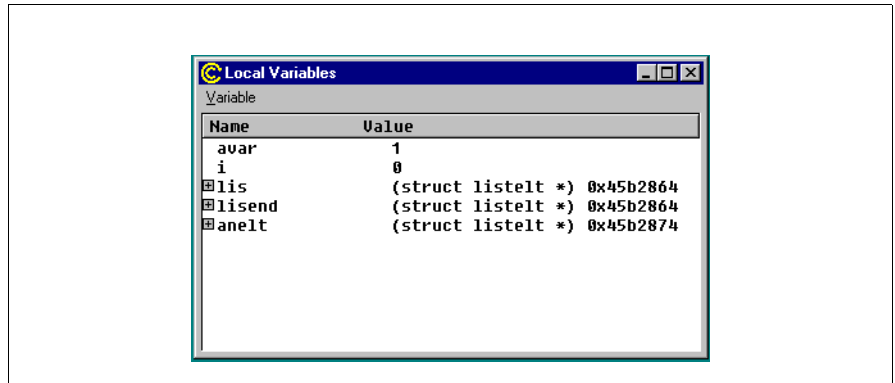


Figure 4-36: Local Variables Window

- Single clicking the mouse with the cursor over a variable selects the variable.
- Double clicking the mouse with the cursor in the **Local Variables** window puts the variable into edit mode.
- Single clicking the mouse with the cursor on the plus sign to the left of a structure variable displays the elements of that structure. See Figure 4-37.

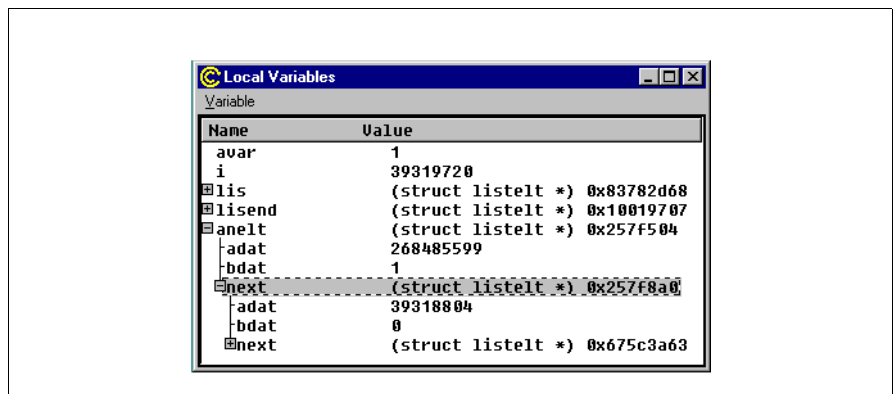


Figure 4-37: Displaying the Elements of a Variable Structure

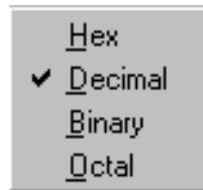
- Single clicking the mouse with the cursor on the minus sign to the left of an open structure closes the display of the structure elements.

Variable Menu

The **Variable** menu of the **Local Variables** window has two options: **Edit** and **Format**.

Edit Allows the value of a selected variable to be edited. Hitting the escape key (Esc) will abort the editing.

Format This menu item brings up another pop-up menu, as shown below, allowing the value of the selected variable to be displayed in the hexadecimal, decimal, binary and octal formats. By default, pointers are displayed in hexadecimal and all other expressions are displayed as decimal.



Breakpoints Window

The **Breakpoints** window displays all breakpoints that are currently set (see Figure 4-38).

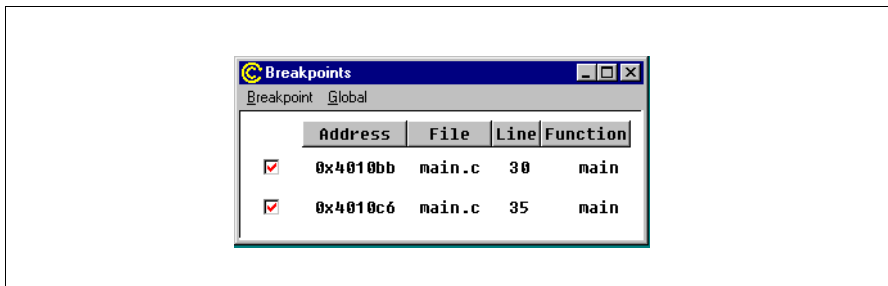


Figure 4-38: Breakpoints Window

- Single clicking with the mouse with the cursor over a check-box for the information displayed for a breakpoint selects that breakpoint.
- Single clicking with the mouse with the cursor over a checked check box of a breakpoint disables the breakpoint. The check disappears and the red square in the **Source Window** turns black.
- Single clicking with the mouse with the cursor over an empty check box of a disabled breakpoint re-enables the breakpoint. The check reappears and the black square in the **Source Window** turns red.

Breakpoint Menu

Figure 4-39 shows the **Breakpoint** menu for the **Breakpoints** window.

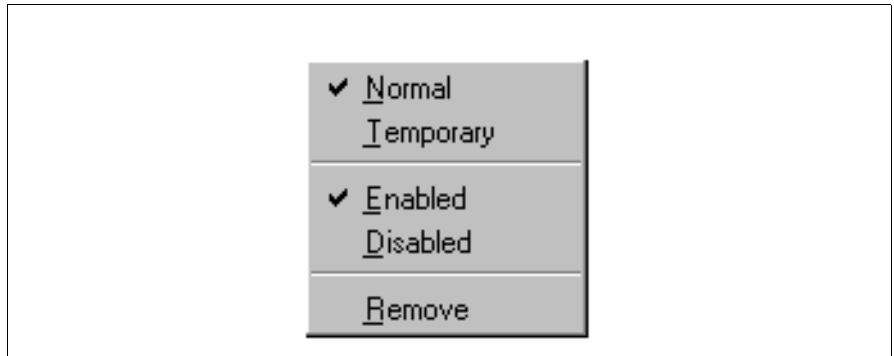


Figure 4-39: Breakpoint Menu

The **Breakpoint** menu options are:

- Normal/Temporary** This pair of menu items toggles between the normal and temporary setting of the 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 red check mark in the check box and the red square in the **Source Window** turn orange. (See Figure 4-40.)

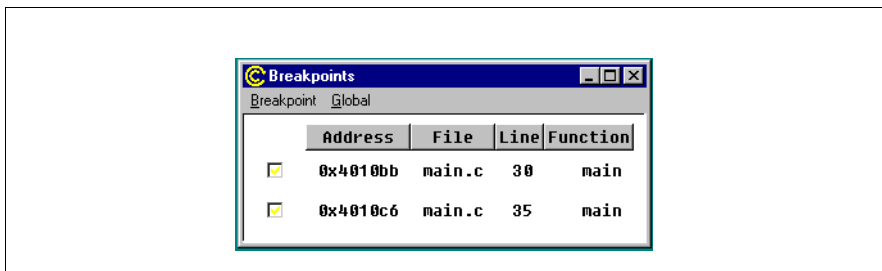


Figure 4-40: Results of Setting Breakpoints

- Enabled/Disabled** This pair of menu items toggles the enabled or disabled state of the selected breakpoint. The single check mark between them shows the state of the selected breakpoint.
- Remove** This menu item removes the selected breakpoint.

Global Menu

Figure 4-41 shows the **Global** menu for the **Breakpoints** window.

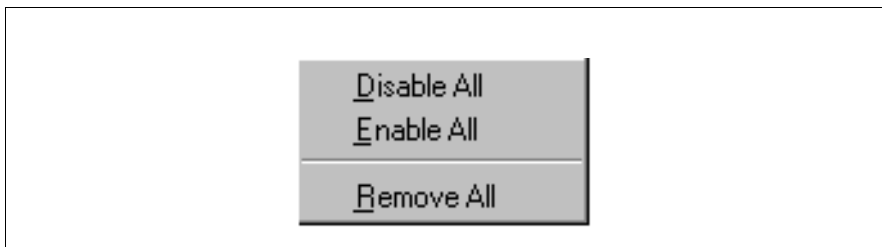


Figure 4-41: Global Menu

The **Global** menu options are:

- Disable All** Disables all breakpoints.
- Enable All** Enables all breakpoints.
- Remove All** Removes all breakpoints.

Console Window

The **Console Window** (see Figure 4-42) contains the command prompt for GDB, the GNUPro debugger, allowing access to the debugger through the command line interface. `(gdb)` is the prompt for the debugger.

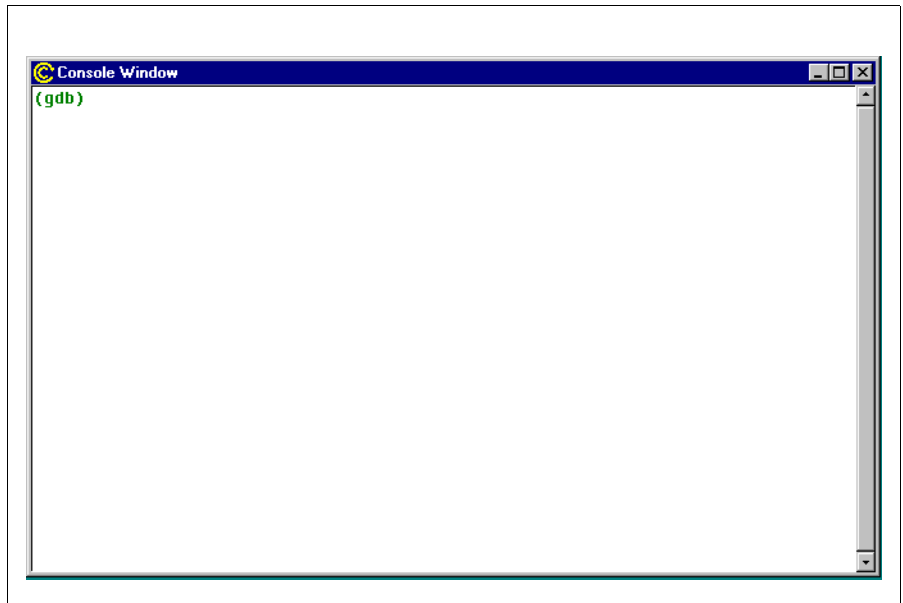


Figure 4-42: Console Window

NOTE: The **Console** window is different from the console window for the Windows operating system (which is known as the Command.com window).

The Function Browser Window

The **Function Browser** window is invoked by clicking on the **Function Browser** menu from the **Source Window**.

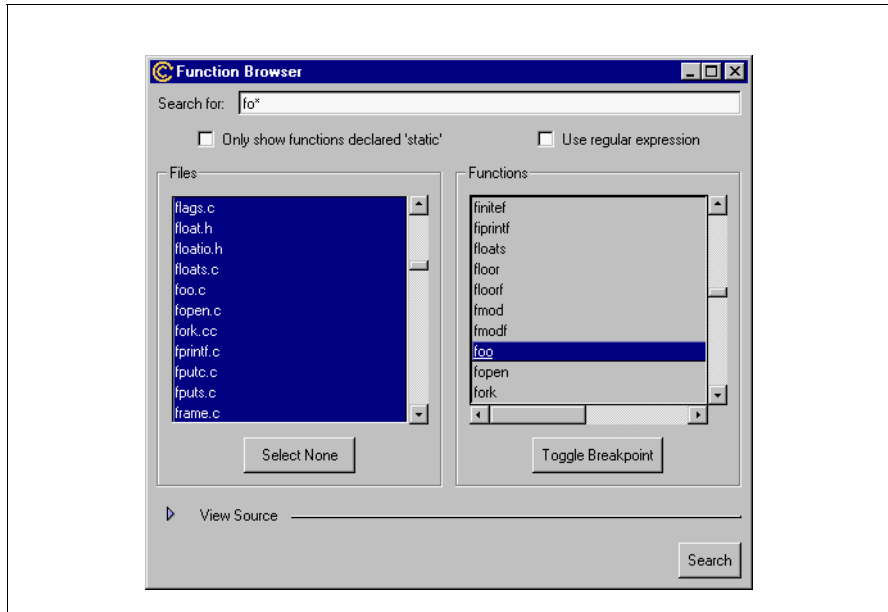


Figure 4-43: Function Browser Window

The **Function Browser** window options are:

- | | |
|--|---|
| Search for: | Text edit field for entering a search expression. |
| Only show functions declared 'static' | Limits listing to static functions. |
| Use regular expression | Makes search routines use regular expression matching. For example, searching for <code>my_func</code> , without using regular expressions, will match <code>my_func_1</code> , not <code>this_is_my_func</code> , while the regular expression, <code>my_func</code> , matches both <code>my_func_1</code> and <code>this_is_my_func</code> regular expressions. |
| Files | Limits the search to the highlighted files. If no files are highlighted, all files are searched. Clicking individual file names selects or deselects that file. |

Select None/ Select All	Toggles between Select All and Select None , switching whenever activated, for selecting all files or none. Useful when searching all files except one or two specific files, or limiting searches to a small group of individually selected files.
Functions	Matches functions in the selected file(s). Right-click on a function to toggle a breakpoint on it.
Toggle Breakpoint	Toggles a breakpoint at all listed functions.
View Source/Hide Source	Toggles to display or hide a source browser. See Figure 4-44.

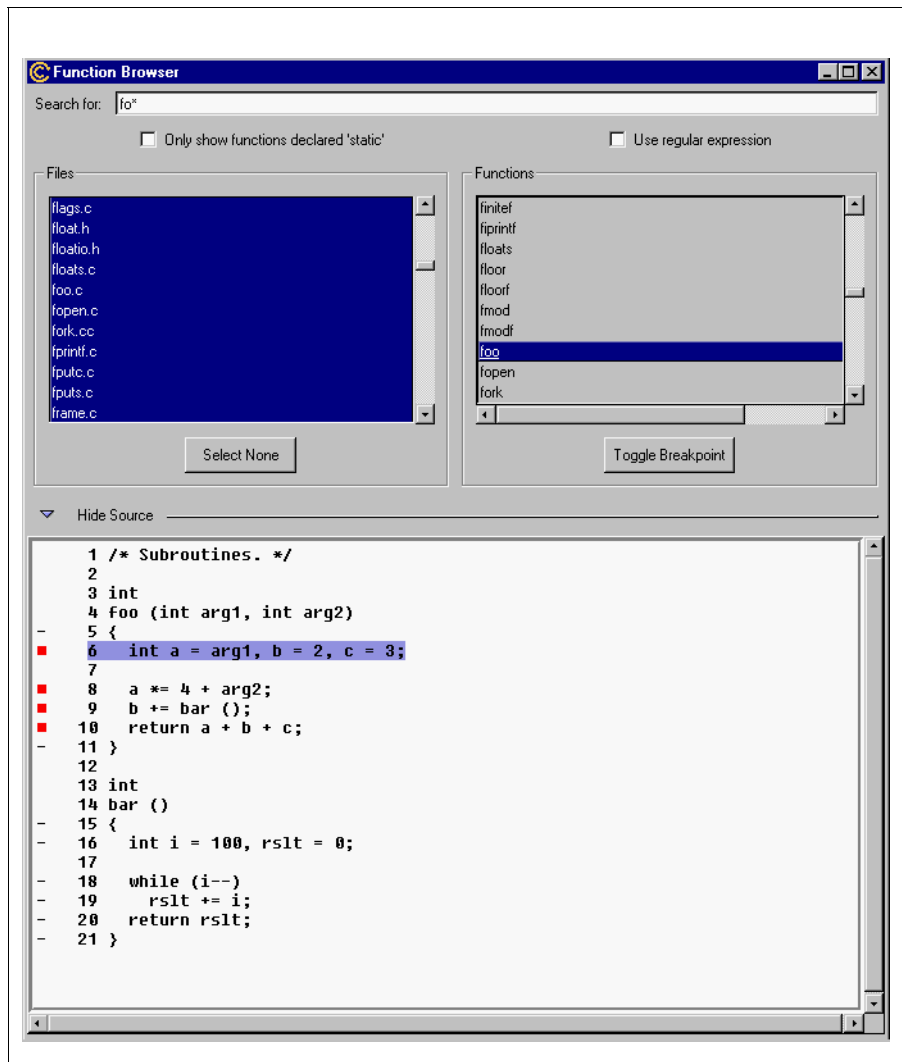


Figure 4-44: Source Browser

Help Window

The **Help** window is invoked by clicking the **Help Topics** menu selection from the **Help** menu of the **Source Window**. The **Help** window offers HTML based navigable help by topic.

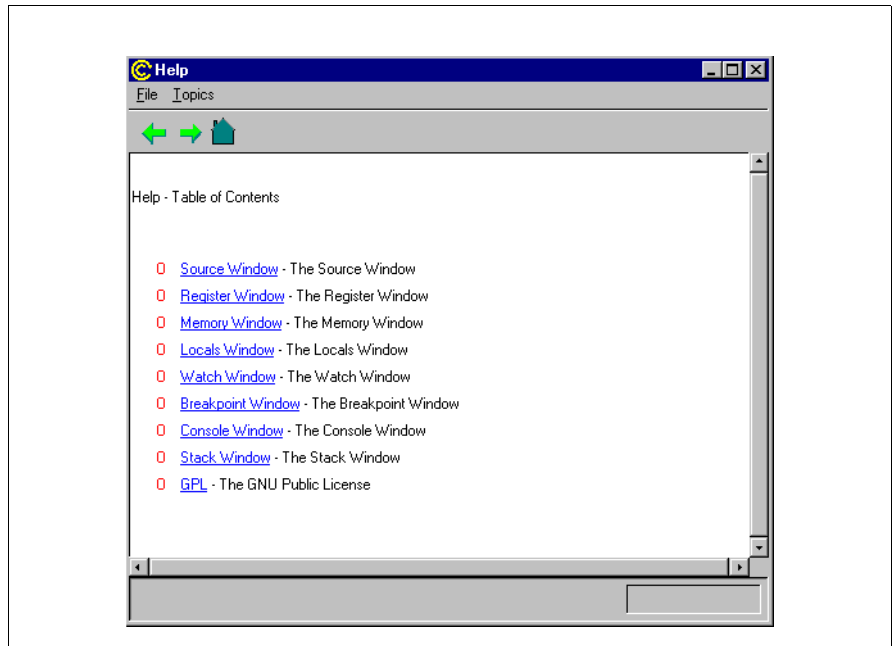


Figure 4-45: Help Window

NOTE: There is currently no Help topic for the **Function Browser** window.

Figure 4-46 shows the **File** menu for the **Help** window.

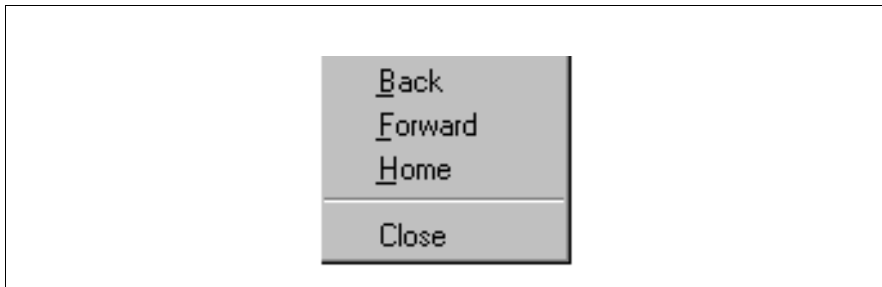


Figure 4-46: Help Window File Menu

Its options are:

- 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 HTML help "Table of Contents" home page.
- Close** Closes the **Help** window.

Topics Menu

Figure 4-47 shows the **Topics** menu for the **Help** window

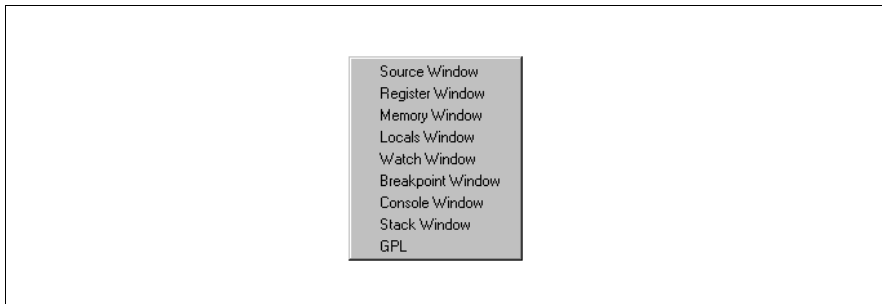


Figure 4-47: Help Topics Menu

Each menu item represents a help topic. When a menu item is selected, the content of the **Help** window changes to reflect the listed topic.

Tutorials for Debugging with Insight

The section contains an example debugging session with step by step procedures for using Insight.

Initializing a Target Executable File

Initializing a target executable file with Insight means opening a specific executable file.

There are two ways to open an executable file in Insight.

- Using the **Open** menu item in the **File** drop-down menu from the **Source Window**.
- Using the following initialization procedure, entering commands at the (gdb) prompt in the **Console** window.
 1. Open the **Console** window, either from the **View** menu, or with the **Console** button (see “Toolbar Buttons” on page 180.).
 2. With the **Console** window active, determine if the target file is in the same directory as Insight. If not, change to the target directory, using the `cd` command.

In our example procedures, the syntax uses the forward slash as the path delimiter on all platforms. Windows, though, requires using two forward slashes after the drive designation.

NOTE: If the source files are not in the same directory as the executable file, use the GDB `dir` command to add a path to them This was not needed in our example.

3. Use the command, file example, to specify the target executable file.

See the following section, “Console Window with Initial Commands” for the results of these procedures.

Console Window with Initial Commands

Figure 4-48 shows the **Console** window with initial commands.

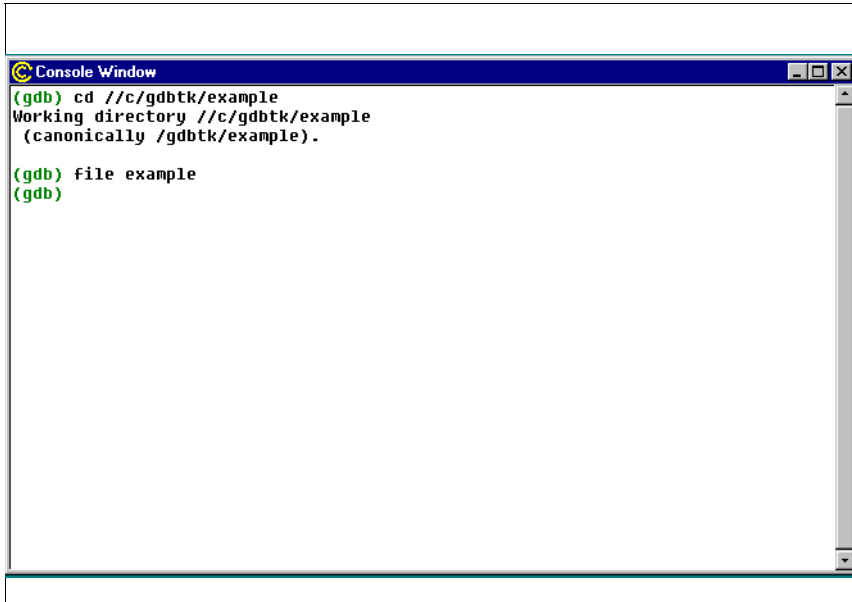


Figure 4-48: Console Window with Initial Commands

Selecting a source file

To select a source file and specify a function within that file, use the following procedure.

1. Select the `foo.c` source file in the file drop-down combo box, at the bottom of the **Source Window**.

Source file and function selection (see Figure 4-49) represents the lower left corner of the **Source Window**, showing the **Source Window's** **File** menu drop-down combo box on the left and the function drop-down combo box

on the right of the window. (See “Below the Horizontal Scroll bar” on page 186.)

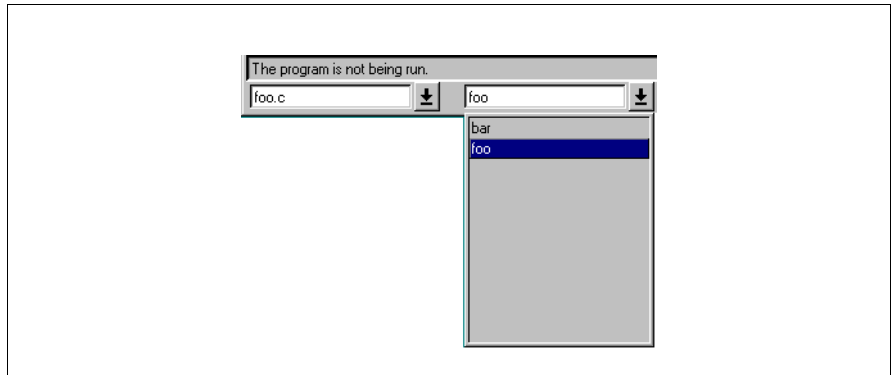


Figure 4-49: Source File and Function Selection

2. Select the function, `foo`, in the function drop-down combo box, at the bottom of the **Source Window**.
3. Now the `foo.c` source file is displayed in the **Source Window** (see Figure 4-50) with a colored bar, indicating the current position. The colored bar is violet, indicating graphically that the program is not running.

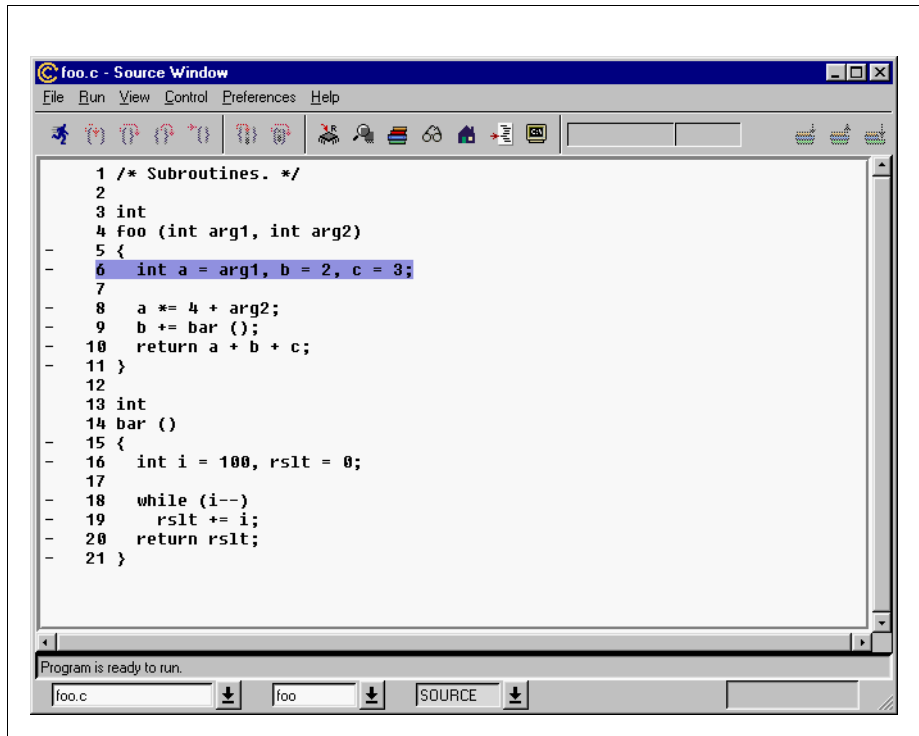


Figure 4-50: Source Window with foo.c Source File

Setting Breakpoints and Viewing Local Variables

A breakpoint can be set at any executable line. 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, having opened the `foo.c` source file, place the cursor 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, signified as a red square.

Note: A second single click on a breakpoint will remove the breakpoint.

- Repeat the process to set breakpoints at lines 8, 9 and 10.
- Open the **Breakpoints** window (see Figure 4-51) by clicking the **Breakpoints** button on the tool bar.

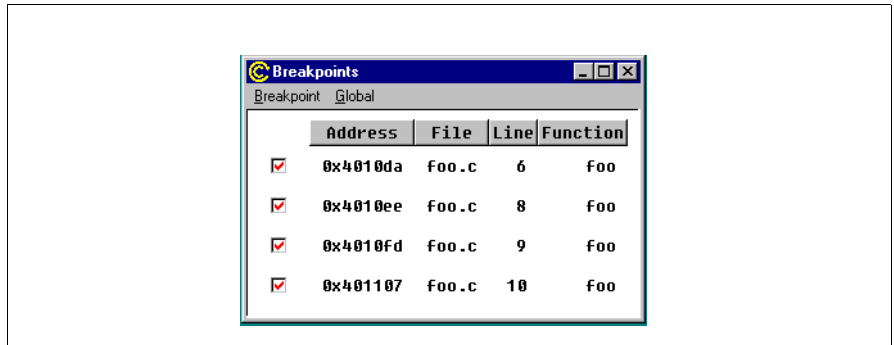


Figure 4-51: Breakpoints Window

- Click the check box for line 6. The red check mark disappears and the red square in the Source Window changes to black. This color change indicates that the breakpoint has been disabled. Re-enable the breakpoint at line 6 by clicking the check box.
- Click the Run button on the tool bar to start the executable (see "Toolbar Buttons" on page 180). The program runs until it hits the first breakpoint

on line 6. The color bar on line 6 is green, indicating that the program is running (see Figure 4-52 and Figure 4-53).

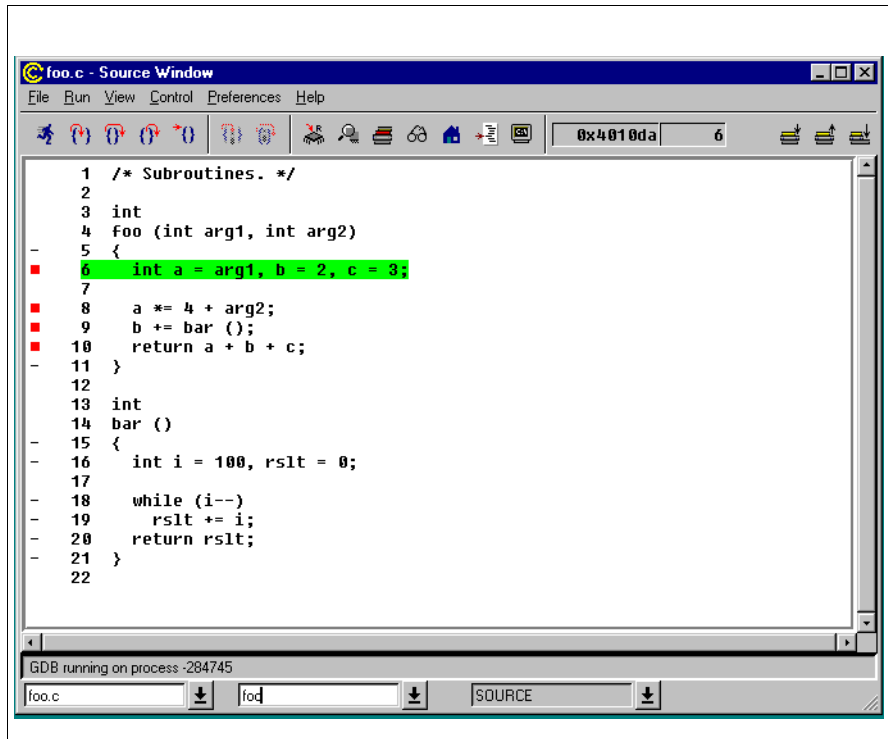


Figure 4-52: Results of Setting Breakpoints

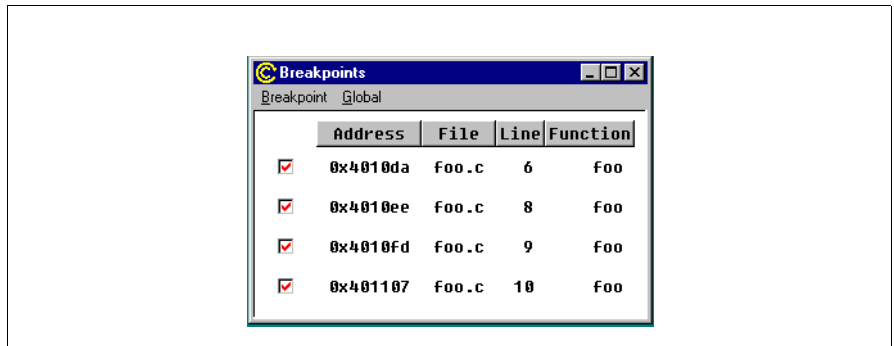


Figure 4-53: Breakpoints Window

- Open the **Local Variables** window, by clicking the **Local Variables** button in the tool bar. The window displays the initial values of the variables.
- Click the **Continue** button in the tool bar (see “Toolbar Buttons” on page 180), to move to the next breakpoint. The variables that have changed value turn blue in the **Local Variables** window (see Figure 4-54).

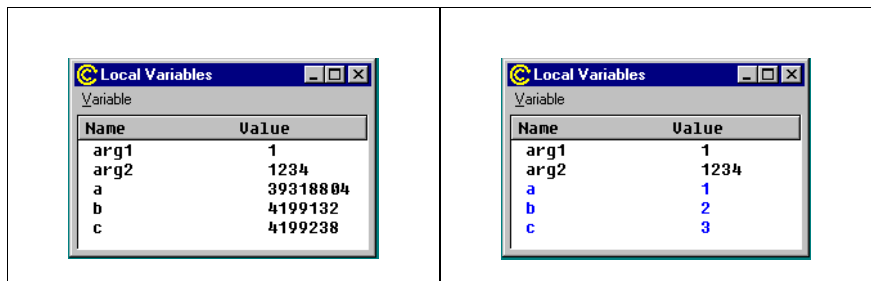


Figure 4-54: Local Variables Window After Setting Breakpoints

- Click the **Continue** button two more times, to step through the next two breakpoints and notice the changing values of the local variables.

Overview

The Simple Kernel Debugger (SKDB) is a machine-level symbolic debugger. This chapter provides an overview of SKDB, instructions on how to install and remove it, and how to start it after a kernel crash, and also lists details of SKDB commands.

SKDB is designed to support debugging of LynxOS kernel internals, primarily device drivers. It allows you to perform the following operations interactively in LynxOS kernel space:

- Setting breakpoints
- Examining memory and registers
- Changing memory contents
- Displaying kernel data structures.

In addition, you can use SKDB to determine the cause of a kernel crash or a kernel panic. LynxOS GDB uses SKDB as the target agent for kernel/device driver debugging.

To use SKDB, the user needs a solid understanding of LynxOS internals, including its memory model, scheduling, interrupt handling, and so forth. SKDB is not designed for user process debugging; use GDB for user process debugging instead.

SKDB has the following characteristics by design:

- SKDB is a tool that you use to debug new device drivers and similar components after the kernel has started and is running fairly stably. It is not intended to be used for porting the LynxOS kernel to a new platform; although SKDB may be still useful for LynxOS kernel porting, it requires a fairly stable kernel and it is not effective in the early stages of the kernel's start-up until the kernel internally installs and initializes SKDB.

- While SKDB is in operation (at its prompt) the entire operating system is paused and no kernel services are available.

Installing/Removing SKDB

In order to use SKDB, install the SKDB module into the kernel. It is possible to install and/or remove SKDB support after initial installation of LynxOS.

Installing SKDB

To install, execute the following:

```
/usr/bin/Install.skdb
```

The installation script will prompt to choose the default SKDB port for SKDB to use when it starts automatically in case of a kernel crash or a kernel panic. SKDB uses this port to *break in* to SKDB with the hot key. The video console, if one exists, always accepts break-ins with the hot key. For more information about starting SKDB automatically, see “Using SKDB” on page 224.

After installation, reboot the system to make SKDB effective.

Removing SKDB

To remove SKDB, execute the following:

```
/usr/bin/Uninstall.skdb
```

Reboot the target system after removal of SKDB.

Using SKDB

SKDB Prompt

Whenever the operating system is in SKDB, it shows an asterisk (*) as its prompt. The entire operating system is paused and no kernel services are available while in SKDB.

Starting SKDB Automatically after a Kernel Crash or Panic

If installed, SKDB is automatically started by a kernel crash, such as kernel memory access fault or a panic situation. In this case, the kernel is usually unable to resume operation, but it is possible to determine the cause and the location of the kernel fault or the panic. For example, the following commands may be useful for analyzing the cause:

- `p` - process/thread table display
- `t` - stack trace display
- `r` - register contents display
- `m` - memory contents display.

Breaking into SKDB with Hot Key

Once SKDB has been installed, it can be invoked by pressing **Shift-Ctrl-Minus** (the “hot key”) on the keyboard of an SKDB-ready port while the operating system is up and running. Some keyboards (mostly video consoles) may use **Ctrl-Minus** instead.

NOTE: To break into SKDB with a hot key from a serial port, the port must have been explicitly opened by a process. Having a `login` process or a dummy process such as `cat` running on the port will suffice for this.

The default hot key combination for SKDB can be changed by using the `z` command within SKDB.

1. At the SKDB prompt, enter the following command:

```
* z
```

SKDB will prompt for a new key combination.

2. Press the desired key combination.

SKDB will prompt for the same key combination for confirmation.

3. Press the same key combination again.

4. To cancel the change, press something else.

The hot key combination is set per port, therefore different key combinations can be set for different ports.

The new hot key combination is not preserved across operating system reboots; it returns to the default combination after each reboot and it needs to be again.

NOTE: To use SKDB for remote kernel debugging with GDB or TotalView, do not change the hot key combination on the serial port. These debuggers have the hot key combination hardcoded.

Kernel Status Display

At each invocation, SKDB prints a line like the following:

```
DELOK:pid.tid@trapcode, slevel, econtext, PSW, PC
checksum
```

DEL	The ASCII Delete character (usually invisible)
OK:	The literal string
pid	Current process ID
tid	Current thread ID
trapcode	Trap code; trap code is the same as the architecture's exception code with the addition of -1 for invocation from keyboard and -2 for a panic situation
slevel	Last slevel; slevel is the kernel preemption level (0: user, 1: kernel, 2: no context switching, 3: no interrupts)
econtext	Econtext address; econtext is the per-thread register stack into which the kernel saves registers
PSW	Processor status word (PSW); called flags or status register in some architectures
PC	Last program counter
checksum	String checksum.

For example, breaking into SKDB with the “hot key” would display something similar to the following:

```
OK:0.0@-1,1,DB0AB19C,000199A0,00000207 C961
```

The above is interpreted that the operating system was running the null process code (process 0) at address 0x207 with context switching enabled when the break-in occurred.

Kernel Status Redisplay

To redisplay the above information, press **Ctrl-B** then **?** and **Return**. This option is currently available on serial terminals only, not on video consoles.

Stack Trace Display

The `t` command displays a traceback or the “history” of nested function calls of a thread within the kernel. One can determine the “path” to the current breakpoint, panic location, or kernel fault location where the kernel entered SKDB. Tracing stops as soon as the stack frame appears to be out of the valid kernel address range.

Give the `t` command the process ID to determine the process’s main thread, or the thread ID with a preceding `-` (minus sign). The default is the current thread.

Verbose Trace Mode

Turning on the verbose trace mode with the `v` command makes the `t` command display the contents of each stack frame as well as the offset values from the frame pointer (or the stack pointer in the case of the PowerPC).

Process, Thread, and Other Displays

The `p` command displays the contents of the kernel’s process table and thread table. The `s` command with options displays the contents of a variety of the kernel’s internal data structures.

Resuming the Kernel

To exit SKDB and resume the operating system, press the **Esc** key; the kernel will continue running until the following occurs:

- Hitting a kernel breakpoint
- Being interrupted by a hot key
- Getting a kernel crash or panic.

As discussed, it may not be possible to resume the kernel if the kernel was in SKDB due to a kernel crash or a kernel panic.

Setting Breakpoints

SKDB can set up to 10 breakpoints in the kernel including device drivers. When the CPU reaches the instruction at a breakpoint, the control is trapped into SKDB. The breakpoints remain set until explicitly unset by the `u` command.

SKDB may refuse to set a breakpoint on some instructions that are critical to its operation. These instructions include those handling the processor status word (PSW) register.



CAUTION! Do not set a breakpoint in the user process space from SKDB. Such a breakpoint will not be recognized by SKDB and thus will cause an unexpected termination of the user process.

Single-Stepping

Pressing `x` and **Return** single-steps the current thread (the thread that caused to enter SKDB.) It is not, however, possible to single-step the following:

- The thread “broken-in” with the hot key
- Some machine instructions that are critical for SKDB’s operation (generally those handling the processor status word (PSW) register)
- A crashed or panicked kernel.



CAUTION! It is not recommended to attempt to trace (setting breakpoints in and/or single-stepping) the “core” portions of the LynxOS kernel, such as those handling context switching and interrupt control, because such an attempt may severely interfere with the LynxOS kernel operation. Also, the instructions that manipulate the processor’s status register cannot be safely single-stepped. Though SKDB detects, warns about, and prevents such an attempt, casual tracing of such critical code may result in an unexpected system freeze.

Disassembly

The `d` command disassembles 10 instructions from the specified address or the current address. The current address is updated to the next text location after each

disassembly. The current address is also updated to the breakpoint or the fault location whenever the kernel stops and reenters SKDB.

NOTE: The current x86 version of SKDB uses the Intel-style syntax of disassembly, not the GNU (AT&T) style.

Setting Watchpoints

Some CPU architectures support hardware debug registers to implement watchpoints. The following LynxOS ports support SKDB watchpoints:

- x86 - up to 4 watchpoints
- PowerPC - up to 1 watchpoint

NOTE: The availability of watchpoint operation for the PowerPC depends on the type of the CPU.

The `B` command can set as many watchpoints as the target CPU architecture allows. The `B` command takes two mandatory arguments—the watchpoint number and the watchpoint location address—plus the following optional arguments:

- Access mode - `r` for read access and `w` for write access; the default is `w`
- Ignore PC addresses - up to 10 text addresses after “!” for the program counter to be ignored for watchpoints hit. This is useful to avoid stopping at known kernel locations where the watchpoint is accessed.

For example, if one wants to catch all write accesses to `currtptr` but does not want to stop at `resched+0x24` which is considered a normal access:

```
* B 1 currtptr w ! resched+0x24
```



CAUTION! “Ignore PC addresses” is implemented by software: all accesses to a watchpoint memory location actually cause CPU exception handling that is captured by SKDB. SKDB examines the cause of the exception and the program counter value to determine whether to resume the kernel silently or to stop and report the hit to the user. This may result in a significant speed penalty if the watchpoint is frequently accessed but ignored.

SKDB uses the virtual address for setting the debug register. Depending on the CPU (MMU) architecture, this may result in watchpoint misses if the page is aliased (mapped at different address locations) and the alias addresses are accessed.

To remove a watchpoint, use the `U` command with the watchpoint number.

SKDB Commands

Table 5-1: SKDB Commands

Command Format	Example	Description
Examine Memory		
<i>hex-addr</i> [<i>size</i>]	* 0xdb100000	Examines (displays) the 4 or <i>size</i> bytes at the location of <i>addr</i>
<i>\$sym</i> ¹ [<i>size</i>]	* \$currpid 10	Examines the 4 or <i>size</i> bytes at the location of <i>sym</i>
m <i>addr</i> [<i>size</i>]	* m currpid 256	Examines <i>size</i> or 64 bytes at <i>addr</i>
T <i>addr</i> [<i>pid</i>]	* T 0xdb100000 9	Translates virtual address <i>addr</i> to physical using <i>pid</i> or current process' mapping
+ [<i>size</i>]	* +	Examines the next 32 or <i>size</i> bytes of memory
- [<i>size</i>]	* - 10	Examines the last 32 or <i>size</i> bytes of memory
Change Memory		
c <i>addr data</i>	* c 0xdb100000 0x200	Stores <i>data</i> as a long word (4 bytes) at <i>addr</i>
Find Symbol		
f <i>addr</i>	* f 0xdb100000	Displays closest symbol with offset to <i>addr</i>
& <i>sym</i>	* &currpid	Displays the address of <i>sym</i>
Display Data Structure		
s st [<i>addr</i> ² <i>tid</i>]	* s st 5	Displays contents of st_entry structure for thread ID <i>tid</i> , address <i>addr</i> , or current thread
s proc [<i>addr</i> ² <i>pid</i>]	* s proc	Displays contents of pentry structure at address <i>addr</i> , for process ID <i>pid</i> , or for the current process

Table 5-1: SKDB Commands(Continued)

Command Format	Example	Description
<code>s pss [addr² pid]</code>	* s pss 0xdb100000	Displays contents of pssentry structure at address <i>addr</i> , for process ID <i>pid</i> , or for the current process
<code>s inode {addr² num}</code>	* s inode 45	Displays contents of <code>inode_entry</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s block {addr² num}</code>	* s block 0xdb100000	Displays contents of <code>buf_entry</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s ihead {addr² num}</code>	* s ihead 30	Displays contents of <code>ihead_entry</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s file {addr² num}</code>	* s file 0xdb100000	Displays contents of <code>file</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s fifo {addr² num}</code>	* s fifo 49	Displays contents of <code>fifo</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s cdev {addr² num}</code>	* s cdev 0xdb100000	Displays contents of <code>cdevsw_entry</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s bdev {addr² num}</code>	* s bdev 0	Displays contents of <code>bdevsw_entry</code> structure at address <i>addr</i> or at index <i>num</i>
<code>s fdentry addr</code>	* s fdentry 0xdb100000	Displays contents of <code>fdentry</code> structure at address <i>addr</i>
<code>s {+ -}</code>	* s +	Displays contents of the next or the last (in memory) data structure of the type being displayed
<code>s {next prev}</code>	* s next	Displays contents of the data structure pointed to by the <code>next</code> or <code>prev</code> (or equivalent) field of the currently-displayed data structure
Stack Trace		
<code>t [pid -tid]</code>	* t -5	Symbolic stack trace of process <i>pid</i> , thread <i>tid</i> , or the current thread

Table 5-1: SKDB Commands(Continued)

Command Format	Example	Description
v	* v	Toggles verbose mode for trace
Display Registers, Processes & Set Priority		
r [<i>pid</i> - <i>tid</i>]	* r	Displays CPU registers of process <i>pid</i> 's main thread, thread <i>tid</i> , or the current thread ³
p [<i>count</i>]	* p 20	Displays process table (all or <i>count</i> lines worth)
P <i>prio tid</i>	* P 15 8	Changes priority of thread <i>tid</i> to <i>prio</i> divided by 2
Breakpoints		
b	* b	Shows all breakpoints set
b <i>num addr</i>	* b 1 0xdb100000	Sets breakpoint <i>num</i> at the memory location <i>addr</i>
u <i>num</i>	* u 5	Unsets breakpoint <i>num</i>
Watchpoints		
B	* B	Shows all watchpoints set
B <i>num addr</i> [<i>r</i> <i>w</i> <i>rw</i>] [! <i>iaddr</i> ...]	* B 1 currtptr w	Sets watchpoint <i>num</i> at the memory location <i>addr</i> for read, write or read/write accesses but ignores accesses by the instruction at <i>iaddr</i>
U <i>num</i>	* U 5	Unsets watchpoint <i>num</i>
Single-Stepping		
x	* x	Single-steps current thread
Disassembly		
d [<i>addr</i>]	* d resched+10	Disassembles at <i>addr</i> or the current PC
Miscellaneous		

Table 5-1: SKDB Commands(Continued)

Command Format	Example	Description
R	* R	Restarts the operating system
h	* h	Displays a description of all available commands
?	* * ?	Same as h
PowerPC Specific		
S	* S	Sees segment register

1. On the PowerPC, a text symbol is preceded by a "." (dot). A symbol without the preceding dot refers to the corresponding TOC entry.
2. The address value must point to a valid table entry.
3. Some architecture may not save all registers upon context switching.

General Notes

Parameter Validation

SKDB performs little validation for command arguments. Although SKDB catches most memory access faults resulting from SKDB commands, improper arguments may result in a system freeze.

Symbol Information

SKDB uses the kernel symbol table that is loaded at the start-up time for symbol lookup. SKDB cannot do interactive symbolic debugging with a stripped kernel.

Address Expressions

SKDB accepts simple address expressions with symbolic notations for most commands that accept memory address parameters. The syntax is as follows:

- Number - hexadecimal if starting with "0x"; octal if starting with "0"; or otherwise decimal

- Symbol - the symbol's absolute virtual address value (note the PowerPC requires a preceding dot for text symbols)
- Register - the CPU register of the *current thread*. The following mnemonics work as common aliases for all architectures: `%pc`, `%fp`, `%sp`. Other register mnemonics depend on the CPU architecture
- Operator - `+` and `-` represent addition and subtraction respectively. Operations are performed left to right without precedence or associativity.

For example, the following sets a breakpoint at the current PC address plus 20 bytes:

```
* b 1 %pc+0x14
```

Default Virtual Address Space

The LynxOS memory model assigns a separate virtual address space to each process (kernel threads belong to process 0 <zero>). Although all processes share the same kernel text, kernel data, and kernel heap in the kernel, each supervisor stack still belongs to its respective process's virtual address space. To access a memory location of a non-current process, use the `T` command to get the memory location's `PHYSBASE` address.

The `PHYSBASE` address is the region of kernel address space where a mirror image of the system's physical memory is mapped (aliased). Since any page that the kernel may access is found in this region and the page is visible to all processes at the same virtual address, `SKDB` uses `PHYSBASE` for quick memory reference in a non-current process's virtual address space.

Remote Debugger Interface Protocol

`SKDB` supports a communication protocol for interfacing with a remote kernel debugger such as LynxOS GDB. For more information on how to debug the LynxOS kernel at the source level, see Chapter 3, See "LynxOS GDB Enhancements" on page 137.

GNU Software License Agreement

GNU General Public License

LynuxWorks, Inc. has derived *Development Support Tools - VOLUME ONE* from the Cygnus Solutions version of the Free Software Foundation GNU documentation. Because this is a work derived from the GNU documentation it falls under the conditions of the GNU public License, and is subject to all the limitations and conditions expressed in that license.

Version 2, June 1991

Copyright© 1989, 1991 - Free Software Foundation, Inc.
59 Temple Place / Suite 330, Boston, MA-- 02111-1307 - USA

Everyone is permitted to copy and distribute verbatim copies of the following documentation of the GNU General Public License, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software- to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all. The precise terms and conditions for copying, distribution and modification follow.

Terms & Conditions for Copying, Distribution and Modification

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification.") Each licensee is addressed as "you."

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the

right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as

distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients- exercise of the rights granted herein.

You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

End of Terms and Conditions

How to Apply these Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line: the program's name and a brief idea of what it does.-

Copyright© 19yy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to:

Free Software Foundation, Inc.,

59 Temple Place - Suite 330

Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like the following example when it starts in an interactive mode:

```
Gnomovision version 69, Copyright© 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for
details type 'show w'. This is free software, and you are
welcome to redistribute it under certain conditions; type
'show c' for details.
```

The `show w` and `show c` hypothetical commands should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they can be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. The following is a sample (when copying, *alter the names*).

```
Yoyodyne, Inc., hereby disclaims all copyright interest
in the program 'Gnomovision' (which makes passes at
compilers) written by James Hacker.
```

```
signature of Ty Coon, 1 April 1989
```

```
Ty Coon, President of Vice
```


This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Contributors to GNU CC

In addition to Richard Stallman, several people have written parts of GNU CC.

- The idea of using RTL and some of the optimization ideas came from the program PO written at the University of Arizona by Jack Davidson and Christopher Fraser. See “Register Allocation and Exhaustive Peephole Optimization,” *Software Practice and Experience* 14 (9), Sept. 1984, pages 857-866.
- Paul Rubin wrote most of the preprocessor.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and the VAX machine description.
- Ted Lemon wrote parts of the RTL reader and printer.
- Jim Wilson implemented loop strength reduction and some other loop optimizations.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Michael Tiemann of Cygnus Solutions wrote the front end for C++, as well as the support for inline functions and instruction scheduling. Also the descriptions of the National Semiconductor 32000 series CPU, the SPARC CPU and part of the Motorola 88000 CPU.
- Gerald Baumgartner added the signature extension to the C++ front-end.
- Jan Stein of the Chalmers Computer Society provided support for GENIX, as well as part of the 32000 machine description.
- Randy Smith finished the Sun™ FPA support.
- Robert Brown implemented the support for Encore 32000 systems.

- David Kashtan of SRI adapted GNU CC to VMS.
- Alex Crain provided changes for the 3b1.
- Greg Satz and Chris Hanson assisted in making GNU CC work on HP-UX for the 9000 series 300.
- William Schelter did most of the work on the Intel 80386 support.
- Christopher Smith did the port for Convex machines.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Alain Lichnewsky ported GNU CC to the MIPS CPU.
- Devon Bowen, Dale Wiles and Kevin Zachmann ported GNU CC to the Tahoe.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Gary Miller ported GNU CC to Charles River Data Systems machines.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination.
- Richard Kenner and Michael Tiemann jointly developed `reorg.c`, the delay slot scheduler.
- Mike Meissner and Tom Wood of Data General finished the port to the Motorola 88000.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- NeXT, Inc. donated the front end that supports the Objective C language.
- James Ivan Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.

- Mike Meissner at the Open Software Foundation finished the port to the MIPS CPU, including adding ECOFF debug support, and worked on the Intel port for the Intel 80386 CPU.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Torbjorn Granlund implemented multiply- and divide-by-constant optimization, improved long long support, and improved leaf function register allocation.
- Mike Stump implemented the support for Elxsi 64 bit CPU.
- John Wehle added the machine description for the Western Electric 32000 processor used in several 3b series machines (no relation to the National Semiconductor 32000 processor).
- Holger Teutsch provided the support for the Clipper CPU.
- Kresten Krab Thorup wrote the run time support for the Objective C language.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits.
- David Edelsohn contributed the changes to RS/6000 port to make it support the PowerPC and POWER2 architectures.
- Steve Chamberlain wrote the support for the Hitachi SH processor.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
- Michael K Gschwind contributed the port to the PDP-11.

Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, “We will donate ten dollars to the Frobnitz project for each disk sold.” Don’t be satisfied with a vague promise, such as “A portion of the profits are donated,” since it doesn’t give a basis for comparison.

Even a precise fraction “of the profits from this disk” is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is \$50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU C compiler contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is “the proper thing to do” when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Copyright© 1994 Free Software Foundation, Inc.

Verbatim copying and redistribution of this section is permitted without royalty; alteration is not permitted.

Protect Your Freedom; Fight “Look And Feel”

This section is a political message from the League for Programming Freedom to the users of GNU CC.

We have included it here because the issue of interface copyright is important to the GNU project.

Apple, Lotus, and now CDC have tried to create a new form of legal monopoly: a copyright on a user interface.

An interface is a kind of language—a set of conventions for communication between two entities, human or machine. Until a few years ago, the law seemed clear: interfaces were outside the domain of copyright, so programmers could program freely and implement whatever interface the users demanded. Imitating de facto standard interfaces, sometimes with improvements, was standard practice in the computer field. These improvements, if accepted by the users, caught on and became the norm; in this way, much progress took place.

Computer users, and most software developers, were happy with this state of affairs. However, large companies such as Apple and Lotus would prefer a different system, one in which they can own interfaces and thereby rid themselves of all serious competitors. They hope that interface copyright will give them, in effect, monopolies on major classes of software.

Other large companies such as IBM and Digital also favor interface monopolies, for the same reason: if languages become property, they expect to own many de facto standard languages. But Apple and Lotus are the ones who have actually sued. Apple’s lawsuit was defeated, for reasons only partly related to the general issue of interface copyright.

Lotus won lawsuits against two small companies, which were thus put out of business. Then they sued Borland; they won in the trial court (no surprise, since it was the same court that had ruled for Lotus twice before), but the decision was reversed by the court of appeals, with help from the League for Programming Freedom in the form of a friend-of-the-court brief. We are now waiting to see if the Supreme Court will hear the case. If it does, the League for Programming Freedom will again submit a brief.

The battle is not over. A company that produced a simulator for a CDC computer was shut down by a copyright lawsuit by CDC, which charged that the simulator infringed the copyright on the manuals for the computer.

If the monopolists get their way, they will hobble the software field:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to design a different way to start, stop, and steer a car.
- Users will be “locked in” to whichever interface they learn; then they will be prisoners of one supplier, who will charge a monopolistic price.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can afford to sue, they can intimidate smaller developers with threats even when they don’t really have a case.

- Interface improvements will come slower, since incremental evolution through creative partial imitation will no longer occur.

If interface monopolies are accepted, other large companies are waiting to grab theirs:

- Adobe is expected to claim a monopoly on the interfaces of various popular application programs, if Lotus ultimately wins the case against Borland.
- Open Computing magazine reported a Microsoft vice president as threatening to sue people who imitate the interface of Windows.

Users invest a great deal of time and money in learning to use computer interfaces. Far more, in fact, than software developers invest in developing and even implementing the interfaces. Whoever can own an interface, has made its users into captives, and misappropriated their investment.

To protect our freedom from monopolies like these, a group of programmers and users have formed a grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose monopolistic practices such as interface copyright and software patents. The League calls for a return to the legal policies of the recent past, in which programmers could program freely. The League is not concerned with free software as an issue, and is not affiliated with the Free Software Foundation.

The League's activities include publicizing the issues, as is being done here, and filing friend-of-the-court briefs on behalf of defendants sued by monopolists.

The League's membership rolls include Donald Knuth, the foremost authority on algorithms, John McCarthy, inventor of Lisp, Marvin Minsky, founder of the MIT Artificial Intelligence lab, Guy L. Steele, Jr., author of well known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are \$42 per year for programmers, managers and professionals; \$10.50 for students; \$21 for others.

Activist members are especially important, but members who have no time to give are also important. Surveys at major ACM conferences have indicated a vast majority of attendees agree with the League on both issues (interface copyrights and software patents). If just ten percent of the programmers who agree with the League join the League, we will probably triumph.

To join, or for more information, phone (617) 243-4091 or write to the League at the following address.

League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139

You can also send electronic mail to: lpf@uunet.uu.net.

In addition to joining the League, here are some suggestions from the League for other things you can do to protect your freedom to write programs:

- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Mention that you are a League member in your '.signature,' and mention the League's e-mail address for inquiries.
- Ask the companies you consider working for or working with to make statements against software monopolies, and give preference to those that do.
- When employers ask you to sign contracts giving them copyright on your work, insist on a clause saying they will not claim the copyright covers imitating the interface.
- When employers ask you to sign contracts giving them patent rights, insist on clauses saying they can use these rights only defensively. Don't rely on company policy, since policies can change at any time; don't rely on an individual executive's private word, since that person may be replaced. Get a commitment just as binding as the commitment they get from you.
- Write to Congress to explain the importance of these issues.

House Subcommittee on Intellectual Property
2137 Rayburn Building
Washington, DC-- 20515

Senate Subcommittee on Patents, Trademarks and Copyrights
United States Senate
Washington, DC-- 20510

(These committees have received lots of mail already; let's give them even more.)

Democracy means nothing if you don't use it. Stand up and be counted!

Index

Symbols

#line directives 87
\$_ variable 84
\$__ variable 84
\$_exitcode variable 84
\$bnum 37
\$cdir string 66
\$cwd string 30
\$num= 82
&(&ref) 94
*address 64
+offset 64
...command-list... 47
@ binary operator 70
@ operator 97
__cp_push_exception library function 43
{type}addr 69
'\$' prefix 82, 83

Numerics

29k architecture 59

A

a struct foo 90
aborting break command 49
Active Targets 112
adder command 116
addr, starting display address 73
add-symbol-file Command 164

add-symbol-file command 109
Alpha architecture 59
Altering Execution 102
altering value variable 103
AMD 29000 family processors 86
Applying terms to new programs 243
argument, for starting GDB 17
Arguments 29
arm 81
Artificial Arrays 70
assert condition 46
Assignment to Variables 103
attach 33
attach process-id 32
auto 81
Automatic Display 74

B

-b bps mode 21
backing up over program 104
Backspace key 23
backtrace 87
backtrace command 58
Backtraces 58
-batch mode 20
--baud option 116
BFD name 113
binary, patching 106
block messages 110
commands 47
break 37, 38
 ...if cond 38
 filename
 function 37
 linenum 37

- function 37
- offset 37
- break *address 38
- break +offset 37
- break command 139
 - aborting 49
- Break Conditions 46
- breakpoint 40
 - command lists 47
 - conditional 43
 - hardware-assisted 39
 - menus 49, 98
 - setting 24, 37, 139
 - settings 34
- breakpoints 23
 - and exceptions 42
 - deleting 44
 - disabling 44
 - enabling 45
 - setting 55
- Breakpoints window 206
- bubble command 25
- bubble option 25

C

- C and C++ Constants 95
- C and C++ Defaults 97
- C and C++ Operators 93
- C and C++ Type and Range Checks 97
- c file 19
- C language 87, 91, 92
- c number 19
- c option 18
- C++ 24, 49, 87
 - code, debugging 96
 - exceptions 43
 - expressions 96
 - language 92
 - program debugging 80
 - references, declared as variables 96
- call expr command 105
- call stack 56
- call_scratch_address variable 105
- calling overloaded functions 96
- Calling Program Functions 105
- canceling execution of function call 105
- cast, use of 71
- catch command 43
- catch exceptions 43
- catchexceptions 98
- cd directory 32
- cd directory mode 20
- changing text commands 129
- Character constants 95
- child process 35
- Choosing Files 18
- Choosing Modes 20
- class class-name 100
- clear 44
- clear filename
 - function 44
 - linenum 44
- clear function 44
- clear linenum 44
- COFF format 108
- comm 115
- command
 - add 116
 - add-symbol-file 109
 - backtrace 58
 - break 139
 - catch 43
 - commandname 117
 - complete args 26
 - completion 23
 - continue 48, 103
 - core-file 106, 112
 - define 116
 - detach 33
 - echo 48
 - editing 12
 - exec-file 106, 107, 112
 - f 123
 - file 19, 106
 - filename 119
 - files 118
 - finish 38, 51
 - frame 57
 - handle 54
 - handle signal keywords... 54
 - help 25, 26
 - help target 113
 - history 13
 - hook-foo 118
 - hooks, user-defined 118
 - if 117
 - info 26

- info frame 89
- info signals 54
- info source 89
- info sources 102
- jobs -l shell 32
- jump 104
- jump *address 104
- jump linespec 103
- kill 33
- line editing 124
- list 63
- list ,last 63
- list first 63
- list function 63
- list linenum 62
- list linespec 63
- load 109
- maint info breakpoints 40
- maint print 102
- M-x 120
- next 51
- nexti 53
- ni 53
- nopass 54
- nostop 54
- output 48
- pass 54
- print 54, 82, 90, 103
- printf 48
- quit 21
- return 105
- return expression 105
- run 28, 29, 32
- section 109
- select-frame 57
- set 26, 103
- set args 30
- set check range 91
- set check type 90
- set complaints 16, 110
- set confirm 17
- set editing 13
- set history expansion 14
- set history filename 13
- set history save 13
- set history size 13
- set input-radix 15
- set language auto 88
- set language local 88
- set listsizecount 63
- set print union 97
- set symbol-reloading 101
- set variable 103
- set verbose 16, 108
- set width 103
- set write 106
- show 26
- show args 30
- show confirm 17
- show input-radix 15
- show language 89
- show listsize 63
- show output-radix 16
- show print 98
- show print union 97
- show range 92
- show type 91
- show verbose 16
- show write 106
- si 53
- signal 104
- silent 48
- source 119
- step 50, 51
- stepi 53
- stop 54
- symbol-file 107
- syntax 22
- target 113
- target remote 116
- thread threadno 55
- tty 32
- u 52
- u location 52
- until 52
- until location 52
- while 117
- x 72, 84
- command lists
 - breakpoint 47
- command-list 48
- commands
 - file management 99
 - for changing text 129
 - for managing targets 113
 - for moving 129
 - GDB 22
 - remotebaud 161
 - target 161
 - GDB add-symbol-file 164

- killing text 126
 - list 64
 - shell 22
 - SKDB 231
 - user-defined 116
 - Commands for manipulating the history 129
 - commands, show history 14
 - compile checks 89
 - Compiling for Debugging 27
 - compiling functions 57
 - complete args command 26
 - condition 46
 - condition bnum expression 46
 - condition, assert 46
 - condition, false 46
 - condition, true 46
 - conditional breakpoint 43
 - configure Options 12
 - confirmation requests, disabling/enabling 17
 - confirmation requests, setting 17
 - Console Window 209
 - contacting LinuxWorks ix
 - Contents i
 - continue argument 47
 - continue command 48, 103
 - Continuing and Stepping 50
 - Contributors to GNU CC 245
 - Controlling GDB 12
 - controlling type checker commands 90
 - convenience variable 40, 74
 - copy permission 27
 - copying terms and conditions 238
 - Copyright Information ii
 - core files 18, 19, 108, 112
 - selecting new 112
 - core-file command 106, 112
 - correcting typos 124
 - correctness, ensuring program 89, 91
 - next 51
 - counter value 58
 - Creating a shared library for debugging 149
 - Ctrl-A 126
 - Ctrl-B 125
 - Ctrl-c system interrupt 58
 - Ctrl-D 125
 - Ctrl-E 126
 - Ctrl-F 125
 - Ctrl-L 126
 - Ctrl-X, Ctrl-R command 128
 - current on or off setting, showing 102
 - current source file, showing name of 101
-
- ## D
- data
 - addresses 41
 - Debug flag, SKDB 160
 - debugger
 - GNU source-level 11
 - Debugging
 - a shared library 140
 - embedded applications remotely 140
 - LynxOS kernels 223
 - POSIX Threads 138
 - Programs with Multiple Processes 35
 - Programs with Multiple Threads 34
 - remote 114
 - Remote Targets 142
 - target, specifying 112
 - with GDB 11
 - Debugging with Insight 215
 - default examining-address 40
 - define command 116
 - Delete 125
 - delete 44
 - delete display dnums... 75
 - Delete key 124
 - Deleting
 - Breakpoints 44
 - detach command 33
 - dev argument 114
 - Device Drivers
 - debugging
 - with GDB 142
 - loading dynamically from GDB 163
 - directory command 66
 - dirname, 66
 - disable display dnums ... 75
 - Disabling Breakpoints 44
 - disassemble command 66, 67
 - display
 - current values of expressions 75
 - display/fmt addr 75
 - display/fmt exp 75
 - display/i \$pc 53
 - displayexp 75
 - displaying
 - core files 106

- current limit 62
- executable files 106
 - the Language 89
- distribution terms and conditions 238
- documents, LynxOS vii
- documents, online vii
- double (virtual) format 85
- down n 59
- down-silently n command 60
- DSU 38, 41

E

- e file 19
- echo command 48
- editing
 - command 12
- Emacs command keystroke sequences 122
- Embedded applications
 - debugging 140
- empty line, significance of 118
- enable display dnms... 75
- enabling breakpoints 45
- encoding algorithm 81
- end 47
- ensuring program correctness 89, 91
- enum enum-tag 100
- Enumerated constants 95
- Environment 29
- environment variable 31
- environment, program 30
- EPROM/ROM code debugging 38
- equivalent variables 97
- error correction 102
- error message, for absent load command 109
- ESC key 24
- Esc key 124
- Event Designators 135
- examine command (x command) 72
- examining
 - data 67
 - memory 72
 - source files 62
 - the stack 56
- exception handlers 61
- exception handling 98
- exceptions 42
- exec-file command 106, 107, 112

- executable file 18
- executable files 112
- execution stack, thread 34
- execution, altering 102
- expression 21, 100, 120
- expressions 96
- expressions, use of regular 98
- extended (raw) format 85
- Extended Remote Targets
 - supported protocols 142

F

- f command 60, 123
- f mode 21
- f, the display format 73
- false condition 46
- file commands 106
- file-management commands 99
- filename 102
 - function 64
 - number 64
- filename command 119
- files, choosing 18
- finish 41
- finish command 38, 51
- fixed address 109
- float parameter 24
- floating point constants 95
- Floating Point Hardware 86
- Floating-point types 93
- foo 48, 81
- format letters supported in output format 72
- forward-searchregex command 65
- frame
 - addr 59
 - args 57
 - command 57
 - n 59
 - pointer register 57
 - selecting 59
- frames
 - stack 57
- free software, funding 247
- fullname mode 21
- function 44, 64
- Function Browser window 210
- function call 56

Function Call in a Multithread Process 170
function invocations 57
function stack, and MIPS machines 61
Funding Free Software 247

G

g++ command 92
GDB 173
 and C 97
 as free software 11
 attaching to a running process 147
 browsing and switching threads 138
 building a kernel for debugging 159
 calling a shared library function by hand
 (PowerPC) 140
 commands 22
 add-symbol-file 164
 compiling `sspp.c` 166
 creating
 a shared library for debugging 149
 current thread 139
 debugging
 device driver/kernel target 142
 the kernel 159
 with threads versus processes 161
 deleting
 breakpoints in shared libraries 152
 description 11
 executing a remote shell from the
 target 169
 features for C++ 98
 files 106
 host machine 115
 installing 12
 installing, `sspp` 167
 interrupting
 the kernel 162
 the thread 171
 loading
 a shared library symbol table 149
 overview 137
 proxy server 165
 Reading and Writing Large Memory
 Blocks 168
 Remote and Extended-remote Targets 142
 requirements 158
 resuming

 after a blocking system call 171
 threads 140
 starting
 gdbserver remotely 144
 the remote target 145
 target command 161
 TCP Port 142
 understanding thread numbers 138
 Using a Serial Line 143
gdb program 17
gdbserver 115
Giving Your Program a Signal 104
GNU 173
 C++ compiler 92
 Emacs 26
 General Public License 237
 History Library 134
 Software License Agreement 237
 source-level debugger 11
gnu 81
-gstabs option 92
-gstabs+ option 93

H

handle command 54
handle signal keywords... command 54
hardware breakpoint registers 39
hardware-assisted breakpoint 39
Having GDB Infer the Source Language 88
hbreak args 38
help command 25, 26
help target commands 113
hex numbers 86
hex, printing number in 71
History Interaction 135
history numbers 82
hook-foo command 118
hook-stop 118

I

id, value of 43
identifier, thread 34
if argument 46
if command 117

- ignore bnum count 47
- ignore count 47
- ignore count, positive 47
- ignore-count argument 50
- info
 - address symbol 99
 - all-registers 84
 - args 61
 - break 40
 - catch 61, 98
 - command 26
 - f 60
 - f addr command 61
 - files 32, 110
 - float 86
 - frame addr command 61
 - frame command 60
 - functions 101
 - functions regexp 101
 - line 77
 - line command 66
 - line linespec 66
 - locals 61
 - program 36
 - registers 84
 - registers regname... 84
 - share 110
 - sharedlibrary 110
 - signals command 54
 - source 100
 - source command 89
 - sources 101
 - sources command 102
 - target 110
 - terminal 32
 - thread 161
 - threads 34
 - threads display 35, 55
 - types 100
 - types regexp 100
 - watchpoints 42
- info display 76
- info frame command 89
- info line command 66
- info variables 101
- info variables regexp 101
- inheritance relationships, printing 98
- init files 119
- initial frame 57
- innermost frame 57, 86

- Insight 173
- inspect 68
- Install.skdb script 160
- Installation 166
- Installing GDB 12
- int 90
- int parameter 24
- integer constants 95
- integer value, storing 103
- Integral types 93
- interrupt 21
- Introduction to Line Editing 124
- invocations, function 57
- Invoking GDB 17

J

- jobs -l shell command 32
- jump *address command 104
- jump command 104
- jump linespec command 103

K

- Kernel
 - debugging with GDB 159
 - interrupting
 - at GDB prompt 162
 - single-stepping 162
- Key Bindings 129
- keyname 129
- keystroke notations 124
- keystroke sequences, Emacs commands 122
- kill command 33
- Killing and Yanking 131
- killing text 126
- killing text commands 126
- Killing the Child Process 33

L

- Language-specific information 87
- LFD (linefeed) key 124
- library for debugging 149
- license 237

- limit 62
- limitations to exception handling 43
- linespec 64
- list .last command 63
- list command 63
- list commands 64
- list first command 63
- list function command 63
- list linenum command 62
- list linespec command 63
- List of Filename Extensions and Languages 87
- load address, specifying 109
- load command 109, 114
- loading
 - device drivers dynamically 163
- Local Variables window 205
- longjmp 40
- longjmp resume 41
- lucid 81
- LynuxWorks, contacting ix
- LynuxWorks, Inc. ii

M

- maint info breakpoints command 40
- maint print commands 102
- maint print msymbols 102
- maint print psymbols 102
- make make-args 22
- Makefile
 - sspp 166
- manipulating history commands 129
- manually setting working language 88
- M-B 126
- Memory window 200
- meta bit 79
- meta digits 127
- META key 24
- M-F 126
- Minicom Terminal Server 167
- MIPS architecture 59
- MIPS Machines and the Function Stack 61
- modes, choosing 20
- modification terms and conditions 238
- Modifiers 136
- moving commands 129
- multiple process debugging 35
- multiple threads 55

- multiple threads, debugging programs with 34
- multi-thread programs 42
- M-x command 120

N

- n 20
- info break 39
- info breakpoints 39
- info watchpoints 39
- n, the repeat count 73
- newprompt 12
- next command 51
- nexti command 53
- ni command 53
 - nn 79
- NO WARRANTY statement, showing 27
- nopass command 54
- noprint command 54
- nostop command 54
- number 64
- Numbers 15
 - thread, understanding GDB 138
- numeric arguments, passing to readline
 - commands 127
- nx mode 20
- nx option 119

O

- object files, replacing 101
- octal constants 95
- off default 97
- offset 64
- on or off setting, showing 102
- online documentation vii
- openlink gdbInvoking_GDB.fm 17
- optimizer 28
- option, -c 18
- option, -se 18
- optional parameters 73
- options, configure 12
- outermost frame 57
- output 120
- output command 48

- overloaded functions, calling 96
- Overloaded symbol names 99
- overloading 49, 98
- Overview
 - GDB 137
 - SKDB 223
- Overview of Range Checking 91
- Overview of Type Checking 90

P

- p/a pointer 78
- parent process 35
- pass command 54
- Patching Programs 106
- path directory 30
- PATH variable 121
- pauses, converting to messages 108
- permission for copying 27
- pointer in decimal, printing 71
- Pointer types 93
- pointer variable, storing 103
- positive ignore count 47
- PowerPC
 - calling a shared library function by hand 140
- print command 54, 82, 90, 103
- print x 82
- printexp command 68
- printf 120
- printf command 48
- printing
 - declared types of objects 98
 - derived types of objects 98
 - names and data types 101
 - source file names 101
 - source lines 62
 - variable names and data types 101
- process ID 18
- process ID, getting 36
- processes 112
- program counter 59
- program functions, calling 105
- Program Variables 69
- programs, patching 106
- prompt string 12
- Protocols
 - proxy 167

- Proxy Server
 - running 165
 - ssp 165
- ps program 35
- pseudo-command 118
- ptt 78
- ptype 100
- ptype exp command 68
- ptype exp ptype 100
- ptype typename 98, 100
- pwd 32

Q

- q mode 20
- quiet mode 20
- quit command 21
- Quitting GDB 21

R

- range checking overview 91
- range checks, C and C++ 97
- raw data format 85
- rbreak regex 39
- rbreakregex 98
- Readline
 - Arguments 127
 - Bare Essentials 125
 - Init File 127
 - Init Syntax 128
 - Interaction 124
 - Killing Commands 126
 - Movement Commands 125
 - vi Mode 134
- readnow option 108, 109
- Reference manuals vii
- regex 39
- Registers window 198
- registers, thread 34
- regular expressions 98
- Remote
 - shell
 - executing from the GDB remote target 169
 - target process, starting from gdbserver 146

- Remote Debugging 114
 - remote serial target 114
 - Remote Targets
 - supported protocols 142
 - repeating list command 63
 - replacing symbol definitions 101
 - Requirements
 - for GDB 158
 - restart, program 56
 - resuming program execution 50
 - Return 22, 23, 33, 63, 71, 74, 75, 82, 83, 84, 85, 104, 107, 109, 113, 124
 - return command 105
 - return expression command 105
 - Return key 23, 33, 48, 124
 - Returning from a Function 105
 - reverse-search regexp command 65
 - run command 28, 29, 32
 - Running Programs Under GDB 27
 - run-time checks 89
-
- S**
- s file 19
 - Scalar types 93
 - Scripts
 - Install.skdb 160
 - se file 19
 - se option 18
 - Searching Source Files 65
 - section command 109
 - select-frame 57
 - select-frame command 57
 - Selecting a Frame 59
 - semantics, thread 34
 - Semaphores 137
 - Serial Line
 - using for GDB 143
 - Serial Ports
 - setting up GDB 161
 - Server
 - GDB proxy 165
 - Minicom Terminal 167
 - Servers
 - starting gdbserver, remotely 144
 - set
 - args command 30
 - check range commands 91
 - check type commands 90
 - command 26, 103
 - complaints command 16
 - confirm command 17
 - demangle-stylestyle 80
 - editing command 13
 - environment 31
 - heuristic-fence-post limit command 62
 - history expansion command 14
 - history filename command 13
 - history save command 13
 - history size command 13
 - input-radix command 15
 - language 87
 - language auto command 88
 - language local command 88
 - listsizecount command 63
 - output-radix command, command, set
 - output-radix 15
 - prompt 12
 - remotebaud command 161
 - rstack_high_address address 86
 - symbol-reloading commands 101
 - variable command 103
 - verbose 16
 - verbose command 108
 - width command 103
 - write command 106
 - x=5 82
 - set command 103
 - set complaints command 110
 - set language 88
 - set print
 - address off 76
 - address on 76
 - array 78
 - asm-demangle 80
 - demangle 80, 98
 - max-symbolic-offsetmax-offset 77
 - null-stop 79
 - object 81
 - pretty on/off 79
 - sevenbit-strings 79
 - symbol-filename 78
 - symbol-filename on/off 77
 - union 79
 - union command 97
 - vtbl 81
 - set print elements number-of-elements 78
 - setprint addressoff 96

- setting
 - breakpoint 24, 37, 139
 - multiple thread breakpoints 55
 - watchpoints 41
 - working language 88
- Setting Up
 - GDB serial ports 161
- Shared Libraries
 - setting GDB breakpoints 152
- Shared Library
 - debugging 140
 - loading for GDB 149
- shared object library symbols, loading 110
- shell command string 22
- Shell Commands 22
- SHELL environment variable 30
- show
 - args command 30
 - command 26
 - confirm command 17
 - convenience 83
 - demangle-style 81
 - directories 66
 - environment 31
 - heuristic-fence-post 62
 - history commands 14
 - input-radix command 15
 - language command 89
 - listsize command 63
 - output-radix command 16
 - paths 31
 - prompt 12
 - range command 92
 - rstack_high_address 86
 - symbol-reloading 102
 - type command 91
 - values 82
 - verbose command 16
 - write command 106
- show print
 - address 77
 - array 78
 - asm-demangle 80
 - commands 98
 - demangle 80
 - elements 78
 - max-symbolic-offset 78
 - object 81
 - pretty 79
 - sevenbit-strings 79
 - symbol-filename 77
 - union 80
 - union command 97
 - vtbl 82
- si 53
- si command 53
- side effects, break conditions 46
- signal 35
- signal command 104
- silent command 48
- silent version of frame 57
- Single-stepping
 - into a shared library function 157
 - the Kernel 162
- SKDB
 - commands 231
 - installing 224
 - overview 223
 - removing 224
 - setting the debug flag 160
- sleep 35
- Source and Machine Code 66
- source command 119
- source directories, specifying 65
- source files, listing 100
- source files, printing names of 101
- source files, searching 65
- source languages, switching between 87
- Source Window 174
- Spacebar key 124
- SPARC architecture 59
- SPARC machine instructions 67
- SPARClite 41
- SPARClite DSU 38
- Special Note formats ix
- Specifying a Debugging Target 112
- specifying single source line 64
- Specifying Source Directories 65
- sspp
 - installing 167
 - makefile 166
 - proxy server 165
- sspp.c
 - compiling 166
- stack frame 52, 56, 61, 69
- Stack Frames 57
- stack frames 85
- stack pointer 59
- stack unwinding 43
- Stack window 197

- stack-frame offset, printing 99
- Standard input and output 29
- Starting
 - gdbserver remotely 144
- Starting Your Program 28
- static variable 70
- step command 50, 51
- step count 51
- stepi command 53
- stepping 50
- stop command 54
- stop, program 55, 56
- Stopping and Continuing 36
- Stopping and Starting Multi-Thread Programs 55
- stopping, before exception handler is called 43
- storing sequences of commands 116
- string table message 111
- struct struct-tag 100
- structure conversion 103
- style key escapes 129
- Supported Languages 92
- suppressing complaints 16
- Switching Between Source Languages 87
- symbol definitions, replacing 101
- symbol information 111
- Symbol Scopes (All Platforms) 157
- symbol(types) 99
- symbol-file command 107
- symbols file 19
- symbols, unusual characters in 99
- Syntax 166
- syntax 87
- syntax, command 22
- systag thread identifier 34

T

- t variable 78
- Tab key 22, 23, 24, 25, 124
- target command 161
- target commands 113
- target machine 115
- target remote command 116
- target, definition for 112
- targets, commands for managing 113
- tbreak args 38
- TCP connections 116

- Technical Support ix
- Terms & conditions, copying, distribution, modification 238
- terms, applying 243
- tbreak args 39
- this, class instance pointer 96
- thread apply 35
- thread number 35
- thread threadno command 55
- Threads
 - current, GDB 139
 - GDB 138
 - resuming GDB 140
- Total/db 173
- trap-generation 41
- true condition 46
- tty command 32
- tty device mode 21
- Type and Range Checking 89
- type checker, commands for controlling 90
- type checking 97
- type checking overview 90
- type checks, C and C++ 97
- type typename 98
- typedef 97
- Typographical Conventions viii
- typos, correcting 124

U

- u command 52
- u location command 52
- u, the unit size 73
- undisplay dnums... 75
- union type 97
- union union-tag 100
- unset environment 31
- until 41
- until command 52
- until location command 52
- up n 59
- up-silently n command 60
- User-Defined Command Hooks 118
- User-Defined Commands 116
- user-defined commands 118
- Using
 - a cast 71
 - GDB Under GNU Emacs 120

- history interactively 134
- target command 141
- Using a Proxy Server 145
- Utilities
 - Install.skdb 160

- x command 40, 68, 72, 84
- x file 19
- x/nfuaddr 72
- x/3iaddr 74
- x86
 - creating a shared library for debugging 148

V

- Variable Settings 128
- variable, altering value of 103
- variable, static 70
- Variables
 - Title
 - BookTitle i
 - Legal ii
 - PartNumber i
- variables, declared by C++ references 96
- variables, equivalent 97
- variables, program 69
- varname 31
- version, showing current running 27
- virtual data format 85
- virtual function tables, print format 98
- void value 83

W

- watch expr 41
- Watch Expressions 202
- Watching Registers 204
- watchpoint 40
- watchpoints, setting 41
- whatis 100
- whatis exp 99
- while command 117
- width variable 103
- word delimiters 99
- Word Designators 135
- Working directory 29
- working language 87
- writing, dump of debugging symbol data 102

X

- x addr 72

Y

- Your Program's Environment 30

