Debugging with GDB

The GNU Source-Level Debugger

HP Ninth Edition, for GDB
June 2000

Richard M. Stallman and Roland H. Pesch

(Send bugs and comments on GDB to bug-gdb@gnu.org.)
Debugging with GDB
TEXinfo 2.257

Published by the Free Software Foundation
59 Temple Place - Suite 330,
Boston, MA 02111-1307 USA
Printed copies are available for $20 each.
ISBN 1-882114-11-6

Summary of GDB

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

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- 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++. For more information, see Section 9.4 [Supported languages], page 83. For more information, see Section 9.4.1 [C and C++], page 84.

Support for Modula-2 and Chill is partial. For information on Modula-2, see Section 9.4.2 [Modula-2], page 89. For information on Chill, see Section 9.4.3 [Chill], page 94.

Debugging Pascal programs which use sets, subranges, file variables, or nested functions does not currently work. GDB does not support entering expressions, printing values, or similar features using Pascal syntax.

GDB can be used to debug programs written in Fortran, although it may be necessary to refer to some variables with a trailing underscore. See Section 9.4.4 [Fortran], page 99.

This version of the manual documents HP WDB 2.0, implemented on HP 9000 systems running Release 10.20, or 11.0 of the HP-UX operating system. HP WDB 2.0 can be used to debug code generated by the HP ANSI C, HP ANSI C++ and HP Fortran compilers as well as the GNU C and C++ compilers. It does not support the debugging of Modula-2 or Chill programs.

Free software

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

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

Contributors to GDB

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

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

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

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

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

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

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

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

Brent Benson of Harris Computer Systems contributed DWARF 2 support.

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

Andreas Schwab contributed M68K Linux support.

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

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

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

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

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

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

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

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

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

Toshiba sponsored the support for the TX39 Mips processor.

Matsushita sponsored the support for the MN10200 and MN10300 processors.

Fujitsu sponsored the support for SPARClite and FR30 processors

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

Michael Snyder added support for tracepoints.

Stu Grossman wrote gdbserver.

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

The following people at the Hewlett-Packard Company contributed support for the PA-RISC 2.0 architecture, HP-UX 10.20, 10.30, and 11.0 (narrow mode), HP's implementation of kernel threads, HP's aC++ compiler, and the terminal user interface: Ben Krepp, Richard Title, John Bishop, Susan Macchia, Kathy Mann, Satish Pai, India Paul, Steve Rehrauer, and Elena Zannoni. Kim Haase and Rosario de la Torre provided HP-specific information in this manual.

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

1 A Sample GDB Session

You can use this manual at your leisure to read all about GDB. However, a handful of commands are enough to get started using the debugger. This chapter illustrates those commands.

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

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

```
$ cd gnu/m4
$ ./m4
define(foo,0000)

foo
0000
define(bar,defn(‘foo’))

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

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

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

```
$ gdb m4

Copyright 1986 - 2000 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 2.0 (based on GDB 4.17-hpwdb-980821)
Wildebeest is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it under
certain conditions.  Type "show copying" to see the conditions.
There is absolutely no warranty for Wildebeest.
Type "show warranty" for details.
Wildebeest was built for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
```

GDB reads only enough symbol data to know where to find the rest when needed; as a result, the first prompt comes up very quickly. We now tell GDB to use a narrower display width than usual, so that examples fit in this manual.

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

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

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

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

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

foo
0000
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(gdb) c
Continuing.

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

baz
0000
```

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

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

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

```
(gdb) quit
```

2 Getting In and Out of GDB

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

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

2.1 Invoking GDB

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

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

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

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

```
gdb program
```

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

```
gdb program core
```

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

```
gdb program 1234
```

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

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

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

```
gdb -silent
```

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

Type

```
gdb -help
```

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

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

2.1.1 Choosing files

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

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

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

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

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

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

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

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

`-command` *file*
`-x` *file* Execute GDB commands from file *file*. See Section 18.3 [Command files], page 183.

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

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

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

`-r`

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

You typically combine the `-mapped` and `-readnow` options in order to build a '.syms' file that contains complete symbol information. (See Section 12.1 [Commands to specify files], page 109, for information on '.syms' files.) A simple GDB invocation to do nothing but build a '.syms' file for future use is:

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

2.1.2 Choosing modes

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

`-nx`

`-n` Do not execute commands found in any initialization files (normally called '.gdbinit', or 'gdb.ini' on PCs). Normally, GDB executes the commands in these files after all the command options and arguments have been processed. See Section 18.3 [Command files], page 183.

`-quiet`

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

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

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

```
Program exited normally.
```

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

`-nowindows`

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

`-windows`

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

`-cd` *directory*

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

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

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

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

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

`-tty` *device*
Run using *device* for your program's standard input and output.

`-tui` Use a Terminal User Interface. For information, use your Web browser to read the file '`TUI.html`', which is usually installed in the directory `/opt/langtools/wdb/doc` on HP-UX systems. Do not use this option if you run GDB from Emacs (see see Chapter 19 [Using GDB under GNU Emacs], page 185).

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

2.1.3 Redirecting HP WDB 2.0 input and output to a file

To redirect HP WDB 2.0 input and output to a file use either of these commands to start the debugger:

```
$ script log1
$ gdb
```

or

```
$ gdb | tee log1
```

2.2 Quitting GDB

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

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

If you have been using GDB to control an attached process or device, you can release it with the `detach` command (see Section 4.7 [Debugging an already-running process], page 25).

2.3 Shell commands

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

`shell` *command string*

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

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

`make` *make-args*

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

3 GDB Commands

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

3.1 Command syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, 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.

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 (typing just ⟨RET⟩) means to repeat the previous command. Certain commands (for example, `run`) will not repeat this way; these are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat.

The `list` and `x` commands, when you repeat them with ⟨RET⟩, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.

GDB can also use ⟨RET⟩ in another way: to partition lengthy output, in a way similar to the common utility `more` (see Section 17.4 [Screen size], page 177). Since it is easy to press one ⟨RET⟩ too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files (see Section 18.3 [Command files], page 183).

3.2 Command completion

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

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

```
(gdb) info bre ⟨TAB⟩
```

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

```
(gdb) info breakpoints
```

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

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

```
(gdb) b make_ ⟨TAB⟩
```

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

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

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

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

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

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

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

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

```
(gdb) b bub ⟨TAB⟩
```

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

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

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

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

3.3 Getting help

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

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

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

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

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

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

```
(gdb) help status
Status inquiries.

List of commands:

info -- Generic command for showing things
```

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

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

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

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

```
complete i
```

results in:

```
if
ignore
info
inspect
```

This is intended for use by GNU Emacs.

In addition to `help`, you can use the GDB commands `info` and `show` to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under `info` and under `show` in the Index point to all the sub-commands. See [Index], page 217.

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

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

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

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

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

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

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

`show warranty`
: Display the GNU "NO WARRANTY" statement, or a warranty, if your version of [No value for "GDB"]v comes with one.

4 Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it.

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

4.1 Compiling for debugging

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.

The HP-UX ANSI C and C++ compilers, as well as 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 might not see that variable—because the compiler might optimize 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!).

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

4.2 Starting your program

`run`
`r`
: Use the `run` command to start your program under GDB. You must first specify the program name (except on VxWorks) with an argument to GDB (see Chapter 2 [Getting In and Out of GDB], page 9), or by using the `file` or `exec-file` command (see Section 12.1 [Commands to specify files], page 109).

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 four categories:

The *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. GDB uses the C shell (`/usr/bin/csh`). See Section 4.3 [Your program's arguments], page 22.

The *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 Section 4.4 [Your program's environment], page 23.

The *working directory.*
: Your program inherits its working directory from GDB. You can set the GDB working directory with the `cd` command in GDB. See Section 4.5 [Your program's working directory], page 24.

The *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 Section 4.6 [Your program's input and output], page 24.

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

When you issue the `run` command, your program begins to execute immediately. See Chapter 5 [Stopping and continuing], page 31, for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. See Chapter 8 [Examining Data], page 63.

If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table, and reads it again. When it does this, GDB tries to retain your current breakpoints.

4.3 Your program's arguments

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

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

`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.

4.4 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 white space or by a system-dependent separator character (':' on Unix, ';' on MS-DOS and MS-Windows). If *directory* is already in the path, it is moved to the front, so it is searched sooner.

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

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

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

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

  For example, this command:

```
set env USER = foo
```

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

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

4.5 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 Section 12.1 [Commands to specify files], page 109.

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

`pwd`
: Print the GDB working directory.

4.6 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.

4.7 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 press ⟨RET⟩ 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 bare-board targets that lack an operating system. You must also have permission to send the process a signal.

When you use `attach`, the debugger finds the program running in the process first by looking in the current working directory, then (if the program is not found) by using the source file search path (see Section 7.3 [Specifying source directories], page 59). You can also use the `file` command to load the program. See Section 12.1 [Commands to Specify Files], page 109.

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. See Section 5.1 [Breakpoints in shared libraries], page 31. 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 press ⟨RET⟩ again after executing the command.

If you exit GDB or use the `run` command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see Section 17.6 [Optional warnings and messages], page 178).

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

4.8 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 file while your program is running.

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

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

4.9 Debugging programs with multiple threads

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

GDB provides these facilities for debugging multi-thread 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

> *Warning:* These facilities are not yet available on every GDB configuration where the operating system supports threads. If your GDB does not support threads, these commands have no effect. For example, a system without thread support shows no output from '`info threads`', and always rejects the `thread` command, like this:
>
> ```
> (gdb) info threads
> (gdb) thread 1
> Thread ID 1 not known.  Use the "info threads" command to
> see the IDs of currently known threads.
> ```

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

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

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

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

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

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

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

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

 For example,

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

On HP-UX systems:

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

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

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

when GDB notices a new thread.

On HP-UX systems, you can control the display of thread creation messages.

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

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

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

Here are commands to get more information about threads:

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

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

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

For example,

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

`thread` *threadno*

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

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

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

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

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

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

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

See Section 5.1.2 [Setting watchpoints], page 35, for information about watchpoints in programs with multiple threads.

4.10 Debugging programs with multiple processes

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

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call to `sleep` in the code which the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the `ps` program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see Section 4.7 [Attach], page 25). From that point on you can debug the child process just like any other process which you attached to.

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

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

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

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

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

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

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

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

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

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

You can use the `catch` command to make GDB stop whenever a `fork`, `vfork`, or `exec` call is made. See Section 5.1.3 [Setting catchpoints], page 37.

5 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 find out why.

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.

5.1 Breakpoints, watchpoints, and catchpoints

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break` command and its variants (see Section 5.1.1 [Setting breakpoints], page 32), to specify the place where your program should stop by line number, function name or exact address in the program.

In HP-UX, SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can set breakpoints in shared libraries before the executable is run. See Section 14.7 [Stopping and starting in shared libraries], page 133.

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see Section 5.1.2 [Setting watchpoints], page 35), 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 Section 8.6 [Automatic display], page 68.

A *catchpoint* is another special breakpoint that stops your program when a certain kind of event occurs, such as the throwing of a C++ exception or the loading of a library. As with watchpoints, you use a different command to set a catchpoint (see Section 5.1.3 [Setting catchpoints], page 37), but aside from that, you can manage a catchpoint like any other breakpoint. (To stop when your program receives a signal, use the `handle` command; see Section 5.3 [Signals], page 47.)

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

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

5.1.1 Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable '`$bpnum`' records the number of the breakpoints you've set most recently; see Section 8.9 [Convenience variables], page 76, for a discussion of what you can do with convenience variables.

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

`break` *function*
: Set a breakpoint at entry to function *function*. When using source languages that permit overloading of symbols, such as C++, *function* may refer to more than one possible place to break. See Section 5.1.8 [Breakpoint menus], page 43, for a discussion of that situation.

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

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

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

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

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

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

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

`break ... if` *cond*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. '`...`' stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.1.6 [Break conditions], page 40, for more information on breakpoint conditions.

`tbreak` *args*
: Set a breakpoint enabled only for one stop. *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See Section 5.1.5 [Disabling breakpoints], page 39.

`hbreak` *args*
: Set a hardware-assisted breakpoint. *args* are the same as for the `break` command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClite DSU and some x86-based targets. These targets will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can take a limited number of breakpoints. For example, on the DSU, only two data breakpoints can be set at a time, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones (see Section 5.1.5 [Disabling], page 39). See Section 5.1.6 [Break conditions], page 40.

`thbreak` *args*
: Set a hardware-assisted breakpoint enabled only for one stop. *args* are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support and some target hardware may not have this support. See Section 5.1.5 [Disabling breakpoints], page 39. See also Section 5.1.6 [Break conditions], page 40.

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

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

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

`info breakpoints` [*n*]
`info break` [*n*]
`info watchpoints` [*n*]

Print a table of all breakpoints, watchpoints, and catchpoints set and not deleted, with the following columns for each breakpoint:

Breakpoint Numbers

Type Breakpoint, watchpoint, or catchpoint.

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

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

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

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

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

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

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

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see Section 5.1.6 [Break conditions], page 40).

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

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

`maint info breakpoints`

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

`breakpoint`
: Normal, explicitly set breakpoint.

`watchpoint`
: Normal, explicitly set watchpoint.

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

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

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

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

`shlib events`
: Shared library events.

5.1.1.1 Simulating Hardware Breakpoints

In some situations hardware breakpoints may not be available (e.g. HP-UX 10.20) or you may want to try something other than a hardware watchpoint.

One thing to try is to make a command line call to `mprotect()` in your program to manually protect the page yourself. See `mprotect(2)` man page for explicit information and limitations.

A situation where you might choose to use this technique would be if you find that an instruction is being rewritten or a single data value is corrupted by an errant pointer.

The limitation is that since one protects the entire page, this becomes unwieldy if a lot changes on that page. You need to make sure that there are few other access to the page.

An example of this technique might be:

```
(gdb) dissass myfunction   ; disassemble functions, get address
0x7aefa260 <myfunction>:stw %r19,-0x1c(%sr0,%sp)
0x7aefa264 <myfunction+4>:ldil L'-0x40000000,%r1
...
End of assembler dump.

(gdb) p mprotect(0x7aefa000,0x1000,0x5)  ; protect one page one
```

Make certain to remove any debugger breakpoints from the page or other reasons the debugger might want to write to the page

5.1.2 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.

Depending on your system, watchpoints may be implemented in software or hardware. GDB does software watchpoints by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution. (But this may still be worth it, to catch errors where you have no clue what part of your program is the culprit.)

On some systems, such as HP-UX 11.x, Linux and some other x86-based targets, GDB includes support for hardware watchpoints, which do not slow down the running of your program.

`watch` *expr*
: Set a watchpoint for an expression. GDB will break when *expr* is written into by the program and its value changes.

`rwatch` *expr*
: Set a watchpoint that will break when watch *expr* is read by the program.

`awatch` *expr*
: Set a watchpoint that will break when *expr* is either read or written into by the program.

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

GDB sets a *hardware watchpoint* if possible. Hardware watchpoints execute very quickly, and the debugger reports a change in value at the exact instruction where the change occurs. If GDB cannot set a hardware watchpoint, it sets a software watchpoint, which executes more slowly and reports the change in value at the next statement, not the instruction, after the change occurs.

When you issue the `watch` command, GDB reports

`Hardware watchpoint` *num*`:` *expr*

if it was able to set a hardware watchpoint.

> *NOTE:* HP-UX does not support `awatch` and `rwatch` but does support hardware watchpoints using page protection.

Currently, the `awatch` and `rwatch` commands can only set hardware watchpoints, because accesses to data that don't change the value of the watched expression cannot be detected without examining every instruction as it is being executed, and GDB does not do that currently. If GDB finds that it is unable to set a hardware breakpoint with the `awatch` or `rwatch` command, it will print a message like this:

```
Expression cannot be implemented with read/access watchpoint.
```

Sometimes, GDB cannot set a hardware watchpoint because the data type of the watched expression is wider than what a hardware watchpoint on the target machine can handle. For example, some systems can only watch regions that are up to 4 bytes wide; on such systems you cannot set hardware watchpoints for an expression that yields a double-precision floating-point number (which is typically 8 bytes wide). As a work-around, it might be possible to break the large region into a series of smaller ones and watch them with separate watchpoints.

If you set too many hardware watchpoints, GDB might be unable to insert all of them when you resume the execution of your program. Since the precise number of active watchpoints is unknown until such time as the program is about to be resumed, GDB might not be able to warn you about this when you set the watchpoints, and the warning will be printed only when the program is resumed:

```
Hardware watchpoint num: Could not insert watchpoint
```

If this happens, delete or disable some of the watchpoints.

The 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 `rwatch` commands, **or** two with `awatch` commands, but you cannot set one watchpoint with one command and the other with a different command. GDB will reject the command if you try to mix watchpoints. Delete or disable unused watchpoint commands before setting new ones.

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

GDB automatically deletes watchpoints that watch local (automatic) variables, or expressions that involve such variables, when they go out of scope, that is, when the execution leaves the block in which these variables were defined. In particular, when the program being debugged terminates, *all* local variables go out of scope, and so only watchpoints that watch global variables remain set. If you rerun the program, you will need to set all such watchpoints again. One way of doing that would be to set a code breakpoint at the entry to the `main` function and when it breaks, set all the watchpoints.

> *Warning:* In multi-thread programs, software watchpoints have only limited usefulness. If GDB creates a software watchpoint, it 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 software watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression. (Hardware watchpoints, in contrast, watch an expression in all threads.)

5.1.3 Setting catchpoints

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

`catch` *event*

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

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

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

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

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

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

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

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

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

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

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

- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program either to abort or to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits. This is the case even if you set a catchpoint for the exception; catchpoints on exceptions are disabled within interactive calls.
- You cannot raise an exception interactively.
- You cannot install an exception handler interactively.

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

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

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

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__raise_exception` (see Section 5.1 [Breakpoints; watchpoints; and exceptions], page 31).

With a conditional breakpoint (see Section 5.1.6 [Break conditions], page 40) that depends on the value of *id*, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

5.1.4 Deleting breakpoints

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

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

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

`clear`
: Delete any breakpoints at the next instruction to be executed in the selected stack frame (see Section 6.3 [Selecting a frame], page 53). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

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

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

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

5.1.5 Disabling breakpoints

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

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

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

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

- Enabled for deletion. The breakpoint stops your program, but immediately after it does so it is deleted permanently. A breakpoint set with the `tbreak` command starts out in this state.

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

`disable` [`breakpoints`] [*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 Section 5.1.1 [Setting breakpoints], page 32), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see Section 5.2 [Continuing and stepping], page 44.)

5.1.6 Break conditions

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

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

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

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached (see Section 5.1.7 [Breakpoint command lists], page 42).

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the `break` command. See Section 5.1.1 [Setting breakpoints], page 32. They can also be changed at any time with the `condition` command.

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

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

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

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

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

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

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

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

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

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

You could achieve the effect of the ignore count with a condition such as '`$foo-- <= 0`' using a debugger convenience variable that is decremented each time. See Section 8.9 [Convenience variables], page 76.

Ignore counts apply to breakpoints, watchpoints, and catchpoints.

5.1.7 Breakpoint command lists

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

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

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

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

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

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

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

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

The commands `echo`, `output`, and `printf` allow you to print precisely controlled output, and are often useful in silent breakpoints. See Section 18.4 [Commands for controlled output], page 184.

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

```
break foo if x>0
commands
silent
```

```
printf "x is %d\n",x
cont
end
```

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

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

5.1.8 Breakpoint menus

Some programming languages (notably C++) permit a single function name to be defined several times, for application in different contexts. This is called *overloading*. When a function name is overloaded, '`break` *function*' is not enough to tell GDB where you want a breakpoint. If you realize this is a problem, you can use something like '`break` *function*(*types*)' to specify which particular version of the function you want. Otherwise, GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt '`>`'. The first two options are always '`[0] cancel`' and '`[1] all`'. Typing *1* sets a breakpoint at each definition of *function*, and typing *0* aborts the `break` command without setting any new breakpoints.

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

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

5.1.9 "Cannot insert breakpoints"

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

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

When this happens, you have three ways to proceed:

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

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

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

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

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

5.2 Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If it stops due to a signal, you may want to use `handle`, or use '`signal 0`' to resume execution. See Section 5.3 [Signals], page 47.)

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

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore` (see Section 5.1.6 [Break conditions], page 40).

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

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

To resume execution at a different place, you can use `return` (see Section 11.4 [Returning from a function], page 107) to go back to the calling function; or `jump` (see Section 11.2 [Continuing at a different address], page 106) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see Section 5.1 [Breakpoints; watchpoints; and catchpoints], page 31) at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

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

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

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

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

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

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

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

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

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

Contrast this with the `return` command (see Section 11.4 [Returning from a function], page 107).

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

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

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

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

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

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop—even though the test in a C `for`-loop is written before the body of the loop. The `until` command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement—not in terms of the actual machine code.

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

`until` *location*
`u` *location* Continue running your program until either the specified location is reached, or the current stack frame returns. *location* is any of the forms of argument acceptable to `break` (see Section 5.1.1 [Setting breakpoints], page 32). 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 do '`display/i $pc`' when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See Section 8.6 [Automatic display], page 68.

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

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

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

5.3 Signals

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

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

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

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

`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 an alias for `info signals`.

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

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

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

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

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

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

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

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

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

You can also use the `signal` command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with '`signal 0`'. See Section 11.3 [Giving your program a signal], page 107.

5.4 Stopping and starting multi-thread programs

When your program has multiple threads (see Section 4.9 [Debugging programs with multiple threads], page 26), you can choose whether to set breakpoints on all threads, or on a particular thread.

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

linespec specifies source lines; there are several ways of writing them, but the effect is always to specify some source line.

Use the qualifier '`thread` *threadno*' with a breakpoint command to specify that you only want GDB to stop the program when a particular thread reaches this breakpoint. *threadno* is one of the numeric thread identifiers assigned by GDB, shown in the first column of the '`info threads`' display.

If you do not specify '`thread` *threadno*' when you set a breakpoint, the breakpoint applies to *all* threads of your program.

You can use the `thread` qualifier on conditional breakpoints as well; in this case, place '`thread` *threadno*' before the breakpoint condition, like this:

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

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

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

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

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

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

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

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

6 Examining the Stack

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

Each time your program performs a function call, information about the call is generated. 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 Section 6.3 [Selecting a frame], page 53.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command (see Section 6.4 [Information about a frame], page 54).

6.1 Stack frames

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

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

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

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

Some compilers provide a way to compile functions so that they operate without stack frames. (For example, the `gcc` option '`-fomit-frame-pointer`' generates functions without a frame.) This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations. If the innermost function invocation has no stack frame, GDB nevertheless regards it as though it had a separate frame, which is numbered zero as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

`frame` *args*
: The `frame` command allows you to move from one stack frame to another, and to print the stack frame you select. *args* may be either the address of the frame or 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`.

6.2 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 typing the system interrupt character, normally *C-c*.

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

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

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

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

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

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

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

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

6.3 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 example:

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

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

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

6.4 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 Section 6.3 [Selecting a frame], page 53.

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

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

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

`info frame` *addr*
`info f` *addr*
: Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the `frame` command. See Section 6.3 [Selecting a frame], page 53.

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

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

`info catch`
: Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit

the associated frame (using the `up`, `down`, or `frame` commands); then type `info catch`. See Section 5.1.3 [Setting catchpoints], page 37.

7 Examining Source Files

GDB can print parts of your program's source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see Section 6.3 [Selecting a frame], page 53), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source; see Chapter 19 [Using GDB under GNU Emacs], page 185.

7.1 Printing source lines

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

Here 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 Chapter 6 [Examining the Stack], page 51), this prints lines centered around that line.

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

By default, GDB prints ten source lines with any of these forms of the `list` command. You can change this 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 with ⟨RET⟩ discards the argument, so it is equivalent to typing just `list`. This is more useful than listing the same lines again. An exception is made for an argument of '`-`'; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the `list` command expects you to supply zero, one or two *linespecs*. Linespecs specify source lines; there are several ways of writing them, but the 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 in the preceding table.

Here 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 example: in C, this is the line with the open brace.

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

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

7.2 Searching source files

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

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

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

7.3 Specifying source directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, 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 that the executable search path is *not* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

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

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

When you start GDB, its source path includes only '`cdir`' and '`cwd`', in that order. 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 '`:`' ('`;`' on MS-DOS and MS-Windows, where '`:`' usually appears as part of absolute file names) or white space. 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 string '`$cdir`' to refer to the compilation directory (if one is recorded), and '`$cwd`' to refer to the current working directory. '`$cwd`' is not the same as '`.`'—the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

`directory`

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

`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 as follows:

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

7.4 Source and machine code

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

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

For example, we can use `info line` to discover the location of the object code for the first line of function `m4_changequote`:

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

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

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

After `info line`, the default address for the `x` command is changed to the starting address of the line, so that '`x/i`' is sufficient to begin examining the machine code (see Section 8.5 [Examining memory], page 67). Also, this address is saved as the value of the convenience variable `$_` (see Section 8.9 [Convenience variables], page 76).

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

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

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

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

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

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

8 Examining Data

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

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

`print`
`print /`*f*
: If you omit *expr*, GDB displays the last value again (from the *value history*; see Section 8.8 [Value history], page 75). This allows you to conveniently inspect the same value in an alternative format.

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

If you are interested in information about types, or about how the fields of a struct or a class are declared, use the `ptype` *exp* command rather than `print`. See Chapter 10 [Examining the Symbol Table], page 101.

8.1 Expressions

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

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

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

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

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

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

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

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

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

8.2 Program variables

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

Variables in expressions are understood in the selected stack frame (see Section 6.3 [Selecting a frame], page 53); they must be either:

- global (or file-static)

or

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

This means that in the function

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

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

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:

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

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

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

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

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

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

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

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

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

To solve such problems, either recompile without optimizations, or use a different debug info format, if the compiler supports several such formats. For example, GCC, the GNU C/C++ compiler usually supports the '`-gstabs`' option. '`-gstabs`' produces debug info in a format that is superior to formats such as COFF. You may be able to use DWARF-2 ('`-gdwarf-2`'), which is also an effective form for debug info. See Section 4.1 [Compiling for Debugging], page 21.

8.3 Artificial arrays

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

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

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

you can print the contents of `array` with

```
p *array@len
```

The left operand of '`@`' must reside in memory. Array values made with '`@`' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see Section 8.8 [Value history], page 75), after printing one out.

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

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

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

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

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see Section 8.9 [Convenience variables], page 76) as a counter in an expression that prints the first interesting value, and then repeat that expression via ⟨RET⟩. For instance, suppose you have an array `dtab` of pointers to structures, and you are interested in the values of a field `fv` in each structure. Here is an example of what you might type:

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

8.4 Output formats

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

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

`x` Regard the bits of the value as an integer, and print the integer in hexadecimal.

`d` Print as integer in signed decimal.

`u` Print as integer in unsigned decimal.

`o` Print as integer in octal.

`t` Print as integer in binary. The letter '`t`' stands for "two".[1]

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

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

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

---

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

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

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

```
p/x $pc
```

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

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

8.5 Examining memory

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

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

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

`b` Bytes.

`h` Halfwords (two bytes).

`w` Words (four bytes). This is the initial default.

`g` Giant words (eight bytes).

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

addr, starting display address

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

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

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

Even though the unit size *u* is ignored for the formats '`s`' and '`i`', you might still want to use a count *n*; for example, '`3i`' specifies that you want to see three machine instructions, including any operands. The command `disassemble` gives an alternative way of inspecting machine instructions; see Section 7.4 [Source and machine code], page 59.

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

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

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

8.6 Automatic display

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

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

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

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

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

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

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

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

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

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

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

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

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

`info display`
: Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

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

8.7 Print settings

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

These settings are useful for debugging programs in any language:

`set print address`
`set print address on`

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

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

`set print address off`

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

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

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

`show print address`

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest 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 example, here GDB shows that a variable `ptt` points at another variable `t`, defined in '`hi2.c`':

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

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

Other settings control how different kinds of objects are printed:

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

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

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

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

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

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

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

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

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

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

This is the default format.

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

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

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

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

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

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

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

For example, given the declarations

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

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

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

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

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

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

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

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

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

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

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

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

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

`hp` Decode based on the HP ANSI C++ (`aCC`) 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*. **Warning:** this setting alone is not sufficient to allow debugging `cfront`-generated executables. GDB would require further enhancement to permit that.

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

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

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

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

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

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

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

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

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

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

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

8.8 Value history

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

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

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

```
p *$.next
```

You can print successive links in the chain by repeating this command—which you can do by just typing ⟨RET⟩.

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

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

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

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

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

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

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

8.9 Convenience variables

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

Convenience variables are prefixed with '`$`'. Any name preceded by '`$`' can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see Section 8.10 [Registers], page 77). (Value history references, in contrast, are *numbers* preceded by '`$`'. See Section 8.8 [Value history], page 75.)

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

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

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

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

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

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

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

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

Repeat that command by typing ⟨RET⟩.

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

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

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

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

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

8.10 Registers

You can refer to machine register contents, in expressions, as variables with names starting with '$'. The names of registers are different for each machine; use `info registers` to 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 detail below, register values are normally relative to the selected stack frame. *regname* may be any register name valid on the machine you are using, with or without the initial '$'.

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

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer[2] with

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

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

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but

[2] This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use `return`; see Section 11.4 [Returning from a function], page 107.

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 Section 6.3 [Selecting a frame], page 53). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with '`frame 0`').

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

8.11 Printing Floating Point Values

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

To print both single- and double-precision values:

```
(gdb) info reg $fr5
fr5     (single precision)     10.1444092
fr5      (double precision)     600000
```

To get the bit pattern, try the following macro:

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

This is what the macro produces:

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

8.12 Floating point hardware

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

`info float`

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

9 Using GDB with Different Languages

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

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

9.1 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, etc.

In addition to the working language, every source file that GDB knows about has its own working language. For some object file formats, the compiler might indicate which language a particular source file is in. However, most of the time GDB infers the language from the name of the file. The language of a source file controls whether C++ names are demangled—this way `backtrace` can show each frame appropriately for its own language. There is no way to set the language of a source file from within GDB, but you can set the language associated with a filename extension. See Section 9.2 [Displaying the language], page 81.

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.

9.1.1 List of filename extensions and languages

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

'`.c`' C source file

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

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

'.ch'
'.c186'
'.c286' CHILL source file.

'.mod' Modula-2 source file

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

In addition, you may set the language associated with a filename extension. See Section 9.2 [Displaying the language], page 81.

9.1.2 Setting the working language

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

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

9.1.3 Having GDB infer the source language

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

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

9.2 Displaying the language

The following commands help you find out which language is the working language, and also what language source files were written in.

`show language`
: Display the current working language. This is the language you can use with commands such as `print` to build and compute expressions that may involve variables in your program.

`info frame`
: Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See Section 6.4 [Information about a frame], page 54, to identify the other information listed here.

`info source`
: Display the source language of this source file. See Chapter 10 [Examining the Symbol Table], page 101, to identify the other information listed here.

In unusual circumstances, you may have source files with extensions not in the standard list. You can then set the extension associated with a language explicitly:

`set extension-language` *.ext language*
: Set source files with extension *.ext* to be assumed to be in the source language *language*. Not valid on Unix systems.

`info extensions`
: List all the filename extensions and the associated languages. Not valid on Unix systems.

9.3 Type and range checking

> *Warning:* 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 you against making seemingly common errors through a series of compile- and run-time checks. These include checking the type of arguments to functions and operators, and making sure mathematical overflows are caught at run time. Checks such as these help to ensure 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 above if you wish. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation via the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your program's source language. See Section 9.4 [Supported languages], page 83, for the default settings of supported languages.

9.3.1 An overview of type checking

Some languages, 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. For example,

```
1 + 2 ⇒ 3
```

but

```
error 1 + 2.3
```

The second example fails because the `CARDINAL` 1 is not type-compatible with the `REAL` 2.3.

For the expressions you use in GDB commands, you can tell the GDB type checker to skip checking; to treat any mismatches as errors and abandon the expression; or to only issue warnings when type mismatches occur, but evaluate the expression anyway. When you choose the last of these, GDB evaluates expressions like the second example above, but also issues a warning.

Even if you turn type checking off, there may be other reasons related to type that prevent GDB from evaluating an expression. For instance, GDB does not know how to add an `int` and a `struct foo`. These particular type errors have nothing to do with the language in use, and usually arise from expressions, such as the one described above, which make little sense to evaluate anyway.

Each language defines to what degree it is strict about type. For instance, 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 Section 9.4 [Supported languages], page 83, for further details on specific languages.

GDB provides some additional commands for controlling the type checker:

`set check type auto`
: Set type checking on or off based on the current working language. See Section 9.4 [Supported languages], page 83, for the default settings for each language.

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

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

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

9.3.2 An overview of range checking

In some languages (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 ⇒ s

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. See Section 9.4 [Supported languages], page 83, 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 Section 9.4 [Supported languages], page 83, for the default settings for each language.

`set check range on`
`set check range off`
: Set range checking on or off, overriding the default setting for the current working language. A warning is issued if the setting does not match the language default. If a range error occurs and range checking is on, then a message is printed and evaluation of the expression is aborted.

`set check range warn`
: Output messages when the GDB range checker detects a range error, but attempt to evaluate the expression anyway. Evaluating the expression may still be impossible for other reasons, such as accessing memory that the process does not own (a typical example from many Unix systems).

`show range`
: Show the current setting of the range checker, and whether or not it is being set automatically by GDB.

9.4 Supported languages

GDB supports C, C++, Fortran, Java, Chill, assembly, and Modula-2.

Section 9.4.4 [Fortran], page 99 for specific information about Fortran.

Some GDB features may be used in expressions regardless of the language you use: the GDB `@` and `::` operators, and the '`{type}addr`' construct (see Section 8.1 [Expressions], page 63) can be used with the constructs of any supported language.

The following 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.

9.4.1 C and C++

Since C and C++ are so closely related, many features of GDB apply to both languages. Whenever this is the case, we discuss those languages together.

The C++ debugging facilities are jointly implemented by the C++ compiler and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with a supported C++ compiler, such as GNU `g++`, or the HP ANSI C++ compiler (`aCC`).

For best results when using GNU C++, use the stabs debugging format. You can select that format explicitly with the `g++` command-line options '`-gstabs`' or '`-gstabs+`'. See section "Options for Debugging Your Program or GNU CC" in *Using GNU CC*, for more information.

9.4.1.1 C and C++ operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types.

For the purposes of C and C++, the following definitions hold:

- *Integral types* include `int` with any of its storage-class specifiers; `char`; `enum`; and, for C++, `bool`.
- *Floating-point types* include `float`, `double`, and `long double` (if supported by the target platform).
- *Pointer types* include all types defined as `(`*type* `*)`.
- *Scalar types* include all of the above.

The following operators are supported. They are listed here in order of increasing precedence:

`,` The comma or sequencing operator. Expressions in a comma-separated list are evaluated from left to right, with the result of the entire expression being the last expression evaluated.

`=` Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.

op`=` Used in an expression of the form *a op*`=` *b*, and translated to *a* `=` *a op b*. *op*`=` and `=` have the same precedence. *op* is any one of the operators `|`, `^`, `&`, `<<`, `>>`, `+`, `-`, `*`, `/`, `%`.

`?:` The ternary operator. *a* `?` *b* `:` *c* can be thought of as: if *a* then *b* else *c*. *a* should be of an integral type.

`||` Logical OR. Defined on integral types.

`&&` Logical AND. Defined on integral types.

`|` Bitwise OR. Defined on integral types.

`^` Bitwise exclusive-OR. Defined on integral types.

`&` Bitwise AND. Defined on integral types.

`==, !=`
: Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<, >, <=, >=`
: Less than, greater than, less than or equal, greater than or equal. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<<, >>`
: left shift, and right shift. Defined on integral types.

`@`
: The GDB "artificial array" operator (see Section 8.1 [Expressions], page 63).

`+, -`
: Addition and subtraction. Defined on integral types, floating-point types and pointer types.

`*, /, %`
: Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.

`++, --`
: Increment and decrement. When appearing before a variable, the operation is performed before the variable is used in an expression; when appearing after it, the variable's value is used before the operation takes place.

`*`
: Pointer dereferencing. Defined on pointer types. Same precedence as `++`.

`&`
: Address operator. Defined on variables. Same precedence as `++`.

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

`-`
: Negative. Defined on integral and floating-point types. Same precedence as `++`.

`!`
: Logical negation. Defined on integral types. Same precedence as `++`.

`~`
: Bitwise complement operator. Defined on integral types. Same precedence as `++`.

`., ->`
: Structure member, and pointer-to-structure member. For convenience, GDB regards the two as equivalent, choosing whether to dereference a pointer based on the stored type information. Defined on `struct` and `union` data.

`.*, ->*`
: Dereferences of pointers to members.

`[]`
: Array indexing. *a*`[`*i*`]` is defined as `*(`*a*`+`*i*`)`. Same precedence as `->`.

`()`
: Function parameter list. Same precedence as `->`.

`::`
: C++ scope resolution operator. Defined on `struct`, `union`, and `class` types.

`::`
: Doubled colons also represent the GDB scope operator (see Section 8.1 [Expressions], page 63). Same precedence as `::`, above.

If an operator is redefined in the user code, GDB usually attempts to invoke the redefined version instead of using the operator's predefined meaning.

9.4.1.2 C and C++ constants

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

- Integer constants are a sequence of digits. Octal constants are specified by a leading '`0`' (i.e. zero), and hexadecimal constants by a leading '`0x`' or '`0X`'. Constants 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. A floating-point constant may also end with a letter '`f`' or '`F`', specifying that the constant should be treated as being of the `float` (as opposed to the default `double`) type; or with a letter '`l`' or '`L`', which specifies a `long double` constant.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (`'`), or a number—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.

9.4.1.3 C++ expressions

GDB expression handling can interpret most C++ expressions.

> *Warning:* GDB can only debug C++ code if you use the proper compiler. Typically, C++ debugging depends on the use of additional debugging information in the symbol table, and thus requires special support. In particular, if your compiler generates a.out, MIPS ECOFF, RS/6000 XCOFF, or ELF with stabs extensions to the symbol table, these facilities are all available. (With GNU CC, you can use the '`-gstabs`' option to request stabs debugging extensions explicitly.) Where the object code format is standard COFF or DWARF in ELF, on the other hand, most of the C++ support in GDB does *not* work.

1. Member function calls are allowed; you can use expressions like

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

2. While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer `this` following the same rules as C++.

3. You can call overloaded functions; GDB resolves the function call to the right definition, with some restrictions. GDB does not perform overload resolution involving user-defined type conversions, calls to constructors, or instantiations of templates that do not exist in the program. It also cannot handle ellipsis argument lists or default arguments.

   It does perform integral conversions and promotions, floating-point promotions, arithmetic conversions, pointer conversions, conversions of class objects to base classes, and standard conversions such as those of functions or arrays to pointers; it requires an exact match on the number of function arguments.

   Overload resolution is always performed, unless you have specified `set overload-resolution off`. See Section 9.4.1.7 [GDB features for C++], page 88.

   You must specify `set overload-resolution off` in order to use an explicit function signature to call an overloaded function, as in

   ```
   p 'foo(char,int)'('x', 13)
   ```

   The GDB command-completion facility can simplify this; see Section 3.2 [Command completion], page 15.
4. GDB understands variables declared as C++ references; you can use them in expressions just as you do in C++ source—they are automatically dereferenced.

   In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The *address* of a reference variable is always shown, unless you have specified '`set print address off`'.
5. GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in an expression like '*scope1*`::`*scope2*`::`*name*'. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see Section 8.2 [Program variables], page 64).

In addition, when used with HP's C++ compiler, GDB supports calling virtual functions correctly, printing out virtual bases of objects, calling functions in a base subobject, casting objects, and invoking user-defined operators.

9.4.1.4 C and C++ defaults

If you allow GDB to set type and range checking automatically, they both default to `off` whenever the working language changes to C or C++. This happens regardless of whether you or GDB selects the working language.

If you allow GDB to set the language automatically, it recognizes source files whose names end with '`.c`', '`.C`', or '`.cc`', etc, and when GDB enters code compiled from one of these files, it sets the working language to C or C++. See Section 9.1.3 [Having GDB infer the source language], page 80, for further details.

9.4.1.5 C and C++ type and range checks

By default, when GDB parses C or C++ expressions, type checking is not used. However, if you turn type checking on, GDB considers two 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.

9.4.1.6 GDB and C

The `set print union` and `show print union` commands apply to the `union` type. When set to '`on`', any `union` that is inside a `struct` or `class` is also printed. Otherwise, it appears as '`{...}`'.

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See Section 8.1 [Expressions], page 63.

9.4.1.7 GDB features for C++

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

breakpoint menus
: When you want a breakpoint in a function whose name is overloaded, GDB breakpoint menus help you specify which function definition you want. See Section 5.1.8 [Breakpoint menus], page 43.

`rbreak` *regex*
: Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. See Section 5.1.1 [Setting breakpoints], page 32.

`catch throw`
`catch catch`
: Debug C++ exception handling using these commands. See Section 5.1.3 [Setting catchpoints], page 37.

`ptype` *typename*
: Print inheritance relationships as well as other information for type *typename*. See Chapter 10 [Examining the Symbol Table], page 101.

`set print demangle`
`show print demangle`
`set print asm-demangle`
`show print asm-demangle`
: Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. See Section 8.7 [Print settings], page 70.

`set print object`
`show print object`
: Choose whether to print derived (actual) or declared types of objects. See Section 8.7 [Print settings], page 70.

`set print vtbl`
`show print vtbl`
: Control the format for printing virtual function tables. See Section 8.7 [Print settings], page 70. (The `vtbl` commands do not work on programs compiled with the HP ANSI C++ compiler (`aCC`).)

`set overload-resolution on`
: Enable overload resolution for C++ expression evaluation. The default is on. For overloaded functions, GDB evaluates the arguments and searches for a function whose signature matches the argument types, using the standard C++ conversion rules (see Section 9.4.1.3 [C++ expressions], page 86, for details). If it cannot find a match, it emits a message.

`set overload-resolution off`
: Disable overload resolution for C++ expression evaluation. For overloaded functions that are not class member functions, GDB chooses the first function of the specified name that it finds in the symbol table, whether or not its arguments are of the correct type. For overloaded functions that are class member functions, GDB searches for a function whose signature *exactly* matches the argument types.

`show overload-resolution`
: Display current overload resolution setting for C++ expression evaluation.

Overloaded symbol names
: You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type *symbol*`(`*types*`)` rather than just *symbol*. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See Section 3.2 [Command completion], page 15, for details on how to do this.

9.4.2 Modula-2

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

9.4.2.1 Operators

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

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

- *Set types* consist of `SET` and `BITSET` types.
- *Boolean types* consist of `BOOLEAN`.

The following operators are supported, and appear in order of increasing precedence:

`,` Function argument or array index separator.

`:=` Assignment. The value of *var* `:=` *value* is *value*.

`<`, `>` Less than, greater than on integral, floating-point, or enumerated types.

`<=`, `>=` Less than, greater than, less than or equal to, greater than or equal to on integral, floating-point and enumerated types, or set inclusion on set types. Same precedence as `<`.

`=`, `<>`, `#` Equality and two ways of expressing inequality, valid on scalar types. Same precedence as `<`. In GDB scripts, only `<>` is available for inequality, since `#` conflicts with the script comment character.

`IN` Set membership. Defined on set types and the types of their members. Same precedence as `<`.

`OR` Boolean disjunction. Defined on boolean types.

`AND`, `&` Boolean conjunction. Defined on boolean types.

`@` The GDB "artificial array" operator (see Section 8.1 [Expressions], page 63).

`+`, `-` Addition and subtraction on integral and floating-point types, or union and difference on set types.

`*` Multiplication on integral and floating-point types, or set intersection on set types.

`/` Division on floating-point types, or symmetric set difference on set types. Same precedence as `*`.

`DIV`, `MOD` Integer division and remainder. Defined on integral types. Same precedence as `*`.

`-` Negative. Defined on `INTEGER` and `REAL` data.

`^` Pointer dereferencing. Defined on pointer types.

`NOT` Boolean negation. Defined on boolean types. Same precedence as `^`.

`.` `RECORD` field selector. Defined on `RECORD` data. Same precedence as `^`.

`[]` Array indexing. Defined on `ARRAY` data. Same precedence as `^`.

`()` Procedure argument list. Defined on `PROCEDURE` objects. Same precedence as `^`.

`::`, `.` GDB and Modula-2 scope operators.

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

9.4.2.2 Built-in functions and procedures

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

a represents an `ARRAY` variable.

c represents a `CHAR` constant or variable.

i represents a variable or constant of integral type.

m represents an identifier that belongs to a set. Generally used in the same function with the metavariable *s*. The type of *s* should be `SET OF` *mtype* (where *mtype* is the type of *m*).

n represents a variable or constant of integral or floating-point type.

r represents a variable or constant of floating-point type.

t represents a type.

v represents a variable.

x represents a variable or constant of one of many types. See the explanation of the function for details.

All Modula-2 built-in procedures also return a result, described below.

`ABS(`*n*`)` Returns the absolute value of *n*.

`CAP(`*c*`)` If *c* is a lower case letter, it returns its upper case equivalent, otherwise it returns its argument.

`CHR(`*i*`)` Returns the character whose ordinal value is *i*.

`DEC(`*v*`)` Decrements the value in the variable *v* by one. Returns the new value.

`DEC(`*v*`,`*i*`)` Decrements the value in the variable *v* by *i*. Returns the new value.

`EXCL(`*m*`,`*s*`)`
Removes the element *m* from the set *s*. Returns the new set.

`FLOAT(`*i*`)` Returns the floating point equivalent of the integer *i*.

`HIGH(`*a*`)` Returns the index of the last member of *a*.

`INC(`*v*`)` Increments the value in the variable *v* by one. Returns the new value.

`INC(`*v*`,`*i*`)` Increments the value in the variable *v* by *i*. Returns the new value.

`INCL(`*m*`,`*s*`)`
Adds the element *m* to the set *s* if it is not already there. Returns the new set.

`MAX(`*t*`)` Returns the maximum value of the type *t*.

`MIN(`*t*`)` Returns the minimum value of the type *t*.

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

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

`SIZE(x)` Returns the size of its argument. *x* can be a variable or a type.

`TRUNC(r)` Returns the integral part of *r*.

`VAL(t,i)` Returns the member of the type *t* whose ordinal value is *i*.

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

9.4.2.3 Constants

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

- Integer constants are simply a sequence of digits. When used in an expression, a constant is interpreted to be type-compatible with the rest of the expression. Hexadecimal integers are specified by a trailing '`H`', and octal integers by a trailing '`B`'.
- Floating point constants appear as a sequence of digits, followed by a decimal point and another sequence of digits. An optional exponent can then be specified, in the form '`E`[`+`|`-`]*nnn*', where '[`+`|`-`]*nnn*' is the desired exponent. All of the digits of the floating point constant must be valid decimal (base 10) digits.
- Character constants consist of a single character enclosed by a pair of like quotes, either single (`'`) or double (`"`). They may also be expressed by their ordinal value (their ASCII value, usually) followed by a '`C`'.
- String constants consist of a sequence of characters enclosed by a pair of like quotes, either single (`'`) or double (`"`). Escape sequences in the style of C are also allowed. See Section 9.4.1.2 [C and C++ constants], page 86, for a brief explanation of escape sequences.
- Enumerated constants consist of an enumerated identifier.
- Boolean constants consist of the identifiers `TRUE` and `FALSE`.
- Pointer constants consist of integral values only.
- Set constants are not yet supported.

9.4.2.4 Modula-2 defaults

If type and range checking are set automatically by GDB, they both default to `on` whenever the working language changes to Modula-2. This happens regardless of whether you or GDB selected the working language.

If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with '`.mod`' sets the working language to Modula-2. See Section 9.1.3 [Having GDB set the language automatically], page 80, for further details.

9.4.2.5 Deviations from standard Modula-2

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

- Unlike in standard Modula-2, pointer constants can be formed by integers. This allows you to modify pointer variables during debugging. (In standard Modula-2, the actual address contained in a pointer variable is hidden from you; it can only be modified through direct assignment to another pointer variable or expression that returned a pointer.)

- C escape sequences can be used in strings and characters to represent non-printable characters. GDB prints out strings with these escape sequences embedded. Single non-printable characters are printed using the '`CHR(`*nnn*`)`' format.
- The assignment operator (`:=`) returns the value of its right-hand argument.
- All built-in procedures both modify *and* return their argument.

9.4.2.6 Modula-2 type and range checks

Warning: in this release, GDB does not yet perform type or range checking.

GDB considers two Modula-2 variables type equivalent if:

- They are of types that have been declared equivalent via a `TYPE` *t1* `=` *t2* statement
- They have been declared on the same line. (Note: This is true of the GNU Modula-2 compiler, but it may not be true of other compilers.)

As long as type checking is enabled, any attempt to combine variables whose types are not equivalent is an error.

Range checking is done on all mathematical operations, assignment, array index bounds, and all built-in functions and procedures.

9.4.2.7 The scope operators :: and .

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

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

where *scope* is the name of a module or a procedure, *module* the name of a module, and *id* is any declared identifier within your program, except another module.

Using the `::` operator makes GDB search the scope specified by *scope* for the identifier *id*. If it is not found in the specified scope, then GDB searches all scopes enclosing the one specified by *scope*.

Using the `.` operator makes GDB search the current scope for the identifier specified by *id* that was imported from the definition module specified by *module*. With this operator, it is an error if the identifier *id* was not imported from definition module *module*, or if *id* is not an identifier in *module*.

9.4.2.8 GDB and Modula-2

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

The `@` operator (see Section 8.1 [Expressions], page 63), while available with any language, is not useful with Modula-2. Its intent is to aid the debugging of *dynamic arrays*, which cannot be created in Modula-2 as they can in C or C++. However, because an address can be specified by an integral constant, the construct '{*type*}*adrexp*' is still useful.

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

9.4.3 Chill

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

This section covers the Chill related topics and the features of GDB which support these topics.

9.4.3.1 How modes are displayed

The Chill Datatype- (Mode) support of GDB is directly related with the functionality of the GNU Chill compiler, and therefore deviates slightly from the standard specification of the Chill language. The provided modes are:

Discrete modes:

- *Integer Modes* which are predefined by BYTE, UBYTE, INT, UINT, LONG, ULONG,
- *Boolean Mode* which is predefined by BOOL,
- *Character Mode* which is predefined by CHAR,
- *Set Mode* which is displayed by the keyword SET.

  ```
  (gdb) ptype x
  type = SET (karli = 10, susi = 20, fritzi = 100)
  ```

  If the type is an unnumbered set the set element values are omitted.
- *Range Mode* which is displayed by type = <basemode> (<lower bound> : <upper bound>), where <lower bound>, <upper bound> can be of any discrete literal expression (e.g. set element names).

Powerset Mode:

A Powerset Mode is displayed by the keyword POWERSET followed by the member mode of the powerset. The member mode can be any discrete mode.

```
(gdb) ptype x
type = POWERSET SET (egon, hugo, otto)
```

Reference Modes:

- *Bound Reference Mode* which is displayed by the keyword REF followed by the mode name to which the reference is bound.
- *Free Reference Mode* which is displayed by the keyword PTR.

Procedure mode

The procedure mode is displayed by type = PROC(<parameter list>) <return mode> EXCEPTIONS (<exception list>). The <parameter list> is a list of the parameter modes. <return mode> indicates the mode of the result of the procedure if any. The exceptionlist lists all possible exceptions which can be raised by the procedure.

Synchronization Modes:

- *Event Mode* which is displayed by EVENT (<event length>), where (<event length>) is optional.
- *Buffer Mode* which is displayed by BUFFER (<buffer length>) <buffer element mode>, where (<buffer length>) is optional.

Timing Modes:

- *Duration Mode* which is predefined by DURATION
- *Absolute Time Mode* which is predefined by TIME

Real Modes:

Real Modes are predefined with REAL and LONG_REAL.

String Modes:

- *Character String Mode* which is displayed by CHARS(<string length>), followed by the keyword VARYING if the String Mode is a varying mode
- *Bit String Mode* which is displayed by BOOLS(<string length>).

Array Mode:

The Array Mode is displayed by the keyword ARRAY(<range>) followed by the element mode (which may in turn be an array mode).

```
(gdb) ptype x
type = ARRAY (1:42)
          ARRAY (1:20)
             SET (karli = 10, susi = 20, fritzi = 100)
```

Structure Mode

The Structure mode is displayed by the keyword STRUCT(<field list>). The <field list> consists of names and modes of fields of the structure. Variant structures have the keyword CASE <field> OF <variant fields> ESAC in their field list. Since the current version of the GNU Chill compiler doesn't implement tag processing (no runtime checks of variant fields, and therefore no debugging info), the output always displays all variant fields.

```
(gdb) ptype str
type = STRUCT (
    as x,
    bs x,
    CASE bs OF
    (karli):
        cs a
    (ott):
        ds x
    ESAC
)
```

9.4.3.2 Locations and their accesses

A location in Chill is an object which can contain values.

A value of a location is generally accessed by the (declared) name of the location. The output conforms to the specification of values in Chill programs. How values are specified is the topic of the next section, Section 9.4.3.3 [Values and their Operations], page 96.

The pseudo-location `RESULT` (or `result`) can be used to display or change the result of a currently-active procedure:

```
set result := EXPR
```

This does the same as the Chill action `RESULT EXPR` (which is not available in GDB).

Values of reference mode locations are printed by `PTR(<hex value>)` in case of a free reference mode, and by `(REF <reference mode>) (<hex-value>)` in case of a bound reference. `<hex value>` represents the address where the reference points to. To access the value of the location referenced by the pointer, use the dereference operator '`->`'.

Values of procedure mode locations are displayed by `{ PROC (<argument modes> ) <return mode> } <address> <name of procedure location>`. `<argument modes>` is a list of modes according to the parameter specification of the procedure and `<address>` shows the address of the entry point.

Substructures of string mode-, array mode- or structure mode-values (e.g. array slices, fields of structure locations) are accessed using certain operations which are described in the next section, Section 9.4.3.3 [Values and their Operations], page 96.

A location value may be interpreted as having a different mode using the location conversion. This mode conversion is written as `<mode name>(<location>)`. The user has to consider that the sizes of the modes have to be equal otherwise an error occurs. Furthermore, no range checking of the location against the destination mode is performed, and therefore the result can be quite confusing.

```
(gdb) print int (s(3 up 4)) XXX TO be filled in !! XXX
```

9.4.3.3 Values and their Operations

Values are used to alter locations, to investigate complex structures in more detail or to filter relevant information out of a large amount of data. There are several (mode dependent) operations defined which enable such investigations. These operations are not only applicable to constant values but also to locations, which can become quite useful when debugging complex structures. During parsing the command line (e.g. evaluating an expression) GDB treats location names as the values behind these locations.

This section describes how values have to be specified and which operations are legal to be used with such values.

`Literal Values`

> Literal values are specified in the same manner as in GNU Chill programs. For detailed specification refer to the GNU Chill implementation Manual chapter 1.5.

`Tuple Values`

> A tuple is specified by `<mode name>[<tuple>]`, where `<mode name>` can be omitted if the mode of the tuple is unambiguous. This unambiguity is derived from the context of a evaluated expression. `<tuple>` can be one of the following:
>
> - *Powerset Tuple*

- *Array Tuple*
- *Structure Tuple* Powerset tuples, array tuples and structure tuples are specified in the same manner as in Chill programs refer to z200/88 chpt 5.2.5.

`String Element Value`
: A string element value is specified by `<string value>(<index>)`, where `<index>` is a integer expression. It delivers a character value which is equivalent to the character indexed by `<index>` in the string.

`String Slice Value`
: A string slice value is specified by `<string value>(<slice spec>)`, where `<slice spec>` can be either a range of integer expressions or specified by `<start expr> up <size>`. `<size>` denotes the number of elements which the slice contains. The delivered value is a string value, which is part of the specified string.

`Array Element Values`
: An array element value is specified by `<array value>(<expr>)` and delivers a array element value of the mode of the specified array.

`Array Slice Values`
: An array slice is specified by `<array value>(<slice spec>)`, where `<slice spec>` can be either a range specified by expressions or by `<start expr> up <size>`. `<size>` denotes the number of arrayelements the slice contains. The delivered value is an array value which is part of the specified array.

`Structure Field Values`
: A structure field value is derived by `<structure value>.<field name>`, where `<field name>` indicates the name of a field specified in the mode definition of the structure. The mode of the delivered value corresponds to this mode definition in the structure definition.

`Procedure Call Value`
: The procedure call value is derived from the return value of the procedure[1].

  Values of duration mode locations are represented by `ULONG` literals.

  Values of time mode locations are represented by `TIME(<secs>:<nsecs>)`.

`Zero-adic Operator Value`
: The zero-adic operator value is derived from the instance value for the current active process.

`Expression Values`
: The value delivered by an expression is the result of the evaluation of the specified expression. If there are error conditions (mode incompatibility, etc.) the evaluation of expressions is aborted with a corresponding error message. Expressions may be parenthesised which causes the evaluation of this expression before any other expression which uses the result of the parenthesised expression. The following operators are supported by GDB:

---

[1] If a procedure call is used for instance in an expression, then this procedure is called with all its side effects. This can lead to confusing results if used carelessly.

`OR, ORIF, XOR`
`AND, ANDIF`
`NOT` Logical operators defined over operands of boolean mode.

`=, /=` Equality and inequality operators defined over all modes.

`>, >=`
`<, <=` Relational operators defined over predefined modes.

`+, -`
`*, /, MOD, REM`
Arithmetic operators defined over predefined modes.

`-` Change sign operator.

`//` String concatenation operator.

`()` String repetition operator.

`->` Referenced location operator which can be used either to take the address of a location (`->loc`), or to dereference a reference location (`loc->`).

`OR, XOR`
`AND`
`NOT` Powerset and bitstring operators.

`>, >=`
`<, <=` Powerset inclusion operators.

`IN` Membership operator.

9.4.3.4 Chill type and range checks

GDB considers two Chill variables mode equivalent if the sizes of the two modes are equal. This rule applies recursively to more complex datatypes which means that complex modes are treated equivalent if all element modes (which also can be complex modes like structures, arrays, etc.) have the same size.

Range checking is done on all mathematical operations, assignment, array index bounds and all built in procedures.

Strong type checks are forced using the GDB command `set check strong`. This enforces strong type and range checks on all operations where Chill constructs are used (expressions, built in functions, etc.) in respect to the semantics as defined in the z.200 language specification.

All checks can be disabled by the GDB command `set check off`.

9.4.3.5 Chill defaults

If type and range checking are set automatically by GDB, they both default to `on` whenever the working language changes to Chill. This happens regardless of whether you or GDB selected the working language.

If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with '`.ch`' sets the working language to Chill. See Section 9.1.3 [Having GDB set the language automatically], page 80, for further details.

9.4.4 Fortran

You can use HP WDB 2.0 to debug programs written in Fortran. HP WDB 2.0 does not distinguish between Fortran 77 and Fortran 90 files.

HP WDB 2.0 includes an option to control case sensitivity.

`case-sensitive [on | off]`
: The default for Fortran is off, for other languages the default is on.

Other supported features are:

- Fortran 90 pointers
- Structures and unions
- Calling functions with integer, logical, real, complex arguments

9.4.4.1 Fortran types

Fortran types supported:

`integer*1, integer*2, integer*4, integer*8`

`logical*1, logical*2, logical*4, logical*8`

`byte`

`real*4, real*8, real*16`

`complex*8, complex*16`

`character*len, character*(*) [len is a user supplied length]`

`arrays`

- allocatable
- assumed-size
- assumed-shape
- adjustable
- automatic
- explicit-shape

Array elements are displays in column-major order Use () for array member access (e.g, arr(i) instead of arr[i]) Use "set print elements" to control the number of elements printed out when specifying a whole array. The default is 200 elements or the number of elements of the array, which ever is smaller.

9.4.4.2 Fortran operators

The following operators are supported. They are listed here in order of increasing precedence:

`=` Assignment

`*, -, *, /`
: Binary operators

`+, -` Unary operators

`**` Exponentiation

`.EQ., =` Equal

`.NE., /=` Not equal, or concatenation

`.LT., <` Less than

`.LE., <=` Less than or equal to

`.GT., >` Greater than

`.GE., >=` Greater than or equal to

`//` Concatenation

`.NOT.` Logical negation

`.AND.` Logical AND

`.OR.` Logical OR

`.EQV.` Logical equivalence

`.NEQV., .XOR.`
Logical non-equivalence

Logical constants are represented as .TRUE. or .FALSE.

GDB includes support for viewing Fortran common blocks.

`info common`
Lists common blocks visible in the current frame.

`info common <common_block_name>`
Lists values of variables in the named common block.

Fortran entry points are supported.

You can set a break point specifying an entry point name.

9.4.4.3 Fortran special issues

Fortran allows `main` to be a non-`main` procedure, therefore to set a breakpoint in the main program, use `break _MAIN_` or `break <`*program_name*`>`.

Do not use `break main` unless it is the name of a non-`main` procedure.

10 Examining the Symbol Table

The commands described in this chapter allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB (see Section 2.1.1 [Choosing files], page 10), or by one of the file-management commands (see Section 12.1 [Commands to specify files], page 109).

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see Section 8.2 [Program variables], page 64). File names are recorded in object files as debugging symbols, but GDB would ordinarily parse a typical file name, like '`foo.c`', as the three words '`foo`' '`.`' '`c`'. To allow GDB to recognize '`foo.c`' as a single symbol, enclose it in single quotes; for example,

```
p 'foo.c'::x
```

looks up the value of `x` in the scope of the file '`foo.c`'.

`info address` *symbol*
: Describe where the data for *symbol* is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.

  Note the contrast with '`print &`*symbol*', which does not work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

`whatis` *expr*
: Print the data type of expression *expr*. *expr* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See Section 8.1 [Expressions], page 63.

`whatis`
: Print the data type of `$`, the last value in the value history.

`ptype` *typename*
: Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form '`class` *class-name*', '`struct` *struct-tag*', '`union` *union-tag*' or '`enum` *enum-tag*'.

`ptype` *expr*
`ptype`
: Print a description of the type of expression *expr*. `ptype` differs from `whatis` by printing a detailed description, instead of just the name of the type.

  For example, for this variable declaration:

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

  the two commands give this output:

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with `whatis`, using `ptype` without an argument refers to the type of `$`, the last value in the value history.

`info types` *regexp*
`info types`
: Print a brief description of all types whose names match *regexp* (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, '`i type value`' gives information on all types in your program whose names include the string `value`, but '`i type ^value$`' gives information only on types whose complete name is `value`.

  This command differs from `ptype` in two ways: first, like `whatis`, it does not print a detailed description; second, it lists all source files where a type is defined.

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

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

`info functions`
: Print the names and data types of all defined functions.

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

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

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

  Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. For example, in VxWorks you can simply recompile a defective object file and keep on running. If you are running on one of these systems, you can allow GDB to reload the symbols for automatically relinked modules:

`set symbol-reloading on`
: Replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.

`set symbol-reloading off`
: Do not replace symbol definitions when 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.

`set opaque-type-resolution on`
: Tell GDB to resolve opaque types. An opaque type is a type declared as a pointer to a `struct`, `class`, or `union`—for example, `struct MyType *`—that is used in one source file although the full declaration of `struct MyType` is in another source file. The default is on.

A change in the setting of this subcommand will not take effect until the next time symbols for a file are loaded.

`set opaque-type-resolution off`
: Tell GDB not to resolve opaque types. In this case, the type is printed as follows:

```
{<no data fields>}
```

`show opaque-type-resolution`
: Show whether opaque types are resolved or not.

`maint print symbols` *filename*
`maint print psymbols` *filename*
`maint print msymbols` *filename*
: Write a dump of debugging symbol data into the file *filename*. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use '`maint print symbols`', GDB includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols GDB has read. You can use the command `info sources` to find out which files these are. If you use '`maint print psymbols`' instead, the dump shows information about symbols that GDB only knows partially—that is, symbols defined in files that GDB has skimmed, but not yet read completely. Finally, '`maint print msymbols`' dumps just the minimal symbol information required for each object file from which GDB has read some symbols. See Section 12.1 [Commands to specify files], page 109, for a discussion of how GDB reads symbols (in the description of `symbol-file`).

11 Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

For example, you can store new values into variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function.

11.1 Assignment to variables

To alter the value of a variable, evaluate an assignment expression. See Section 8.1 [Expressions], page 63. For example,

```
print x=4
```

stores the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4). See Chapter 9 [Using GDB with Different Languages], page 79, for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the expression's value is not printed and is not put in the value history (see Section 8.8 [Value history], page 75). 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 just `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
```

Because the `set` command has many subcommands that can conflict with the names of program variables, it is a good idea to use the `set variable` command instead of just `set`. For example, if your program has a variable `g`, you run into problems if you try to set a new value with just '`set g=4`', because GDB has the command `set gnutarget`, abbreviated `set g`:

```
(gdb) whatis g
type = double
(gdb) p g
$1 = 1
(gdb) set g=4
(gdb) p g
$2 = 1
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/smith/cc_progs/a.out
"/home/smith/cc_progs/a.out": can't open to read symbols: Invalid bfd target.
(gdb) show g
The current BFD target is "=4".
```

The program variable `g` did not change, and you silently set the `gnutarget` to an invalid value. In order to set the variable `g`, use

```
(gdb) set var g=4
```

GDB allows more implicit conversions in assignments than C; you can freely store an integer value into a pointer variable or vice versa, and you can convert any structure to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the '`{...}`' construct to generate a value of specified type at a specified address (see Section 8.1 [Expressions], page 63). For example, `{int}0x83040` refers to memory location `0x83040` as an integer (which implies a certain size and representation in memory), and

```
set {int}0x83040 = 4
```

stores the value 4 into that memory location.

11.2 Continuing at a different address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the `continue` command. You can 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 Section 7.1 [Printing source lines], page 57, for a description of the different forms of *linespec*. It is common practice to use the `tbreak` command in conjunction with `jump`. See Section 5.1.1 [Setting breakpoints], page 32.

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*.

On many systems, 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 Section 5.2 [Continuing and stepping], page 44.

The most common occasion to use the `jump` command is to back up—perhaps with more breakpoints set—over a portion of a program that has already executed, in order to examine its execution in more detail.

11.3 Giving your program a signal

`signal` *signal*

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

Alternatively, if *signal* is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinary see the signal when resumed with the `continue` command; '`signal 0`' causes it to resume without a signal.

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

Invoking the `signal` command is not the same as invoking the `kill` utility from the shell. Sending a signal with `kill` causes GDB to decide what to do with the signal depending on the signal handling tables (see Section 5.3 [Signals], page 47). The `signal` command passes the signal directly to your program.

11.4 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 Section 6.3 [Selecting a frame], page 53), and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The `return` command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the `finish` command (see Section 5.2 [Continuing and stepping], page 44) resumes execution until the selected stack frame returns naturally.

11.5 Calling program functions

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

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

For the A29K, a user-controlled variable `call_scratch_address`, specifies the location of a scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack does not work in systems that have separate instruction and data spaces.

11.6 Patching programs

By default, GDB opens the file containing your program's executable code (or the core file) 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` command) after changing `set write`, for your new setting to take effect.

`show write`
: Display whether executable files and core files are opened for writing as well as reading.

12 GDB Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, you must also tell GDB the name of the core dump file.

12.1 Commands to specify files

You 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 Chapter 2 [Getting In and Out of GDB], page 9).

Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify a file you want to use. In these situations the GDB commands to specify new files are useful.

`file` *filename*
: Use *filename* as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the `run` command. If you do not specify a directory and the file is not found in the GDB working directory, GDB uses the environment variable `PATH` as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the `path` command.

  On systems with memory-mapped files, an auxiliary file '*filename*`.syms`' may hold symbol table information for *filename*. If so, GDB maps in the symbol table from '*filename*`.syms`', starting up more quickly. See the descriptions of the file options '`-mapped`' and '`-readnow`' (available on the command line, and with the commands `file`, `symbol-file`, or `add-symbol-file`, described below), for more information.

`file`
: `file` with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file` [*filename*]
: Specify that the program to be run (but not the symbol table) is found in *filename*. GDB searches the environment variable `PATH` if necessary to locate your program. Omitting *filename* means to discard information on the executable file.

`symbol-file` [*filename*]
: Read symbol table information from file *filename*. `PATH` is searched when necessary. Use the `file` command to get both symbol table and program to run from the same file.

  `symbol-file` with no argument clears out GDB information on 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 press ⟨RET⟩ again after executing it once.

When GDB is configured for a particular environment, it understands debugging information in whatever format is the standard generated for that environment; you may use either a GNU compiler, or other compilers that adhere to the local conventions.

For most kinds of object files, with the exception of old SVR3 systems using COFF, the `symbol-file` command does not normally read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, as they are needed.

The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The `set verbose` command can turn these pauses into messages if desired. See Section 17.6 [Optional warnings and messages], page 178.)

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. Note that "stabs-in-COFF" still does the two-stage strategy, since the debug info is actually in stabs format.

`symbol-file` *filename* [`-readnow`] [`-mapped`]
`file` *filename* [`-readnow`] [`-mapped`]

You can override the GDB two-stage strategy for reading symbol tables by using the '`-readnow`' option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available.

If memory-mapped files are available on your system (usually not Unix systems, through the `mmap` system call, you can use another option, '`-mapped`', to cause GDB to write the symbols for your program into a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file (if the program has not changed), rather than spending time reading the symbol table from the executable program. Using the '`-mapped`' option has the same effect as starting GDB with the '`-mapped`' command-line option.

You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program.

The auxiliary symbol file for a program called *myprog* is called '*myprog*`.syms`'. Once this file exists (so long as it is newer than the corresponding executable), GDB always attempts to use it when you debug *myprog*; no special options or commands are needed.

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

`core-file` [*filename*]

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

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

Note that the core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the `kill` command (see Section 4.8 [Killing the child process], page 25).

`add-symbol-file` *filename address*
`add-symbol-file` *filename address* [`-readnow`] [`-mapped`]
`add-symbol-file` *filename address data_address bss_address*
`add-symbol-file` *filename* -T*section address*

The `add-symbol-file` command reads additional symbol table information from the file *filename*. You would use this command when *filename* has been dynamically loaded (by some other means) into the program that is running. *address* should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify up to three addresses, in which case they are taken to be the addresses of the text, data, and bss segments respectively. For complicated cases, you can specify an arbitrary number of -T*section address* pairs, to give an explicit section name and base address for that section. You can specify any *address* as an expression.

The symbol table of the file *filename* is added to the symbol table originally read with the `symbol-file` command. You can use the `add-symbol-file` command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `symbol-file` command.

`add-symbol-file` does not repeat if you press ⟨RET⟩ after using it.

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

`add-shared-symbol-file`

The `add-shared-symbol-file` command can be used only under Harris' CXUX operating system for the Motorola 88k. GDB automatically looks for shared libraries, however if GDB does not find yours, you can run `add-shared-symbol-file`. It takes no arguments.

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

`info files`
`info target`

`info files` and `info target` are synonymous; both print the current target (see Chapter 13 [Specifying a Debugging Target], page 115), including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command `help target` lists all possible targets rather than current ones.

All file-specifying commands allow both absolute and relative file names as arguments. GDB always converts the file name to an absolute file name and remembers it that way.

GDB supports HP-UX, SunOS, SVR4, Irix 5, and IBM RS/6000 shared libraries.

GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library, however—unless you are debugging a core file).

On HP-UX, if the program loads a library explicitly, GDB automatically loads the symbols at the time of the `shl_load` call. See Section 5.1 [Stopping and starting in shared libraries], page 31, for more information.

`info share`
`info sharedlibrary`
: Print the names of the shared libraries which are currently loaded.

`sharedlibrary` *regex*
`share` *regex*
: Load shared object library symbols for files matching a Unix regular expression. As with files loaded automatically, it only loads shared libraries required by your program for a core file or after typing `run`. If *regex* is omitted all shared libraries required by your program are loaded.

On HP-UX systems, GDB detects the loading of a shared library and automatically reads in symbols from the newly loaded library, up to a threshold that is initially set but that you can modify if you wish.

Beyond that threshold, symbols from shared libraries must be explicitly loaded. To load these symbols, use the command `sharedlibrary` *filename*. The base address of the shared library is determined automatically by GDB and need not be specified.

To display or set the threshold, use the commands:

`set auto-solib-add` *threshold*
: Set the autoloading size threshold, in megabytes. If *threshold* is nonzero, symbols from all shared object libraries will be loaded automatically when the inferior begins execution or when the dynamic linker informs GDB that a new library has been loaded, until the symbol table of the program and libraries exceeds this threshold. Otherwise, symbols must be loaded manually, using the `sharedlibrary` command. The default threshold is 100 megabytes.

`show auto-solib-add`
: Display the current autoloading size threshold, in megabytes.

12.2 Errors reading symbol files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the `set complaints` command (see Section 17.6 [Optional warnings and messages], page 178).

The messages currently printed, and their meanings, include:

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

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

`block at` *address* `out of order`
: The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so.

  GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine what source file is affected by specifying `set verbose on`. See Section 17.6 [Optional warnings and messages], page 178.)

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

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

`bad string table offset in symbol` *n*
: Symbol number *n* contains a pointer into the string table which is larger than the size of the string table.

  GDB circumvents the problem by considering the symbol to have the name `foo`, which may cause other problems if many symbols end up with this name.

`unknown symbol type 0x`*nn*
: The symbol information contains new data types that GDB does not yet know how to read. `0x`*nn* is the symbol type of the uncomprehended information, in hexadecimal.

  GDB circumvents the error by ignoring this symbol information. This usually allows you to debug your program, though certain symbols are not accessible. If you encounter such a problem and feel like debugging it, you can debug `gdb` with itself, breakpoint on `complain`, then go up to the function `read_dbx_symtab` and examine `*bufp` to see the symbol.

`stub type has NULL name`
: GDB could not find the full definition for a struct or class.

`const/volatile indicator missing (ok if using g++ v1.x), got...`
: The symbol information for a C++ member function is missing some information that recent versions of the compiler should have output for it.

`info mismatch between compiler and debugger`
: GDB could not parse a type specification output by the compiler.

13 Specifying a Debugging Target

A *target* is the execution environment occupied by your program.

Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the `file` or `core` commands. For HP-UX specific information, See Section 14.2 [HP-UX Targets], page 120. When you need more flexibility—for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection—you can use the `target` command to specify one of the target types configured for GDB (see Section 13.2 [Commands for managing targets], page 115).

13.1 Active targets

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

For example, if you execute '`gdb a.out`', then the executable file `a.out` is the only active target. If you designate a core file as well—presumably from a prior run that crashed and core dumped—then GDB has two active targets and uses them in tandem, looking first in the core file 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 Section 12.1 [Commands to specify files], page 109). To specify as a target a process that is already running, use the `attach` command (see Section 4.7 [Debugging an already-running process], page 25).

13.2 Commands for managing targets

`target` *type parameters*

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

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

The `target` command does not repeat if you press ⟨RET⟩ again after executing the command.

`help target`
: Displays the names of all targets available. To display targets currently selected, use either `info target` or `info files` (see Section 12.1 [Commands to specify files], page 109).

`help target` *name*
: Describe a particular target, including any parameters necessary to select it.

`set gnutarget` *args*
: GDB uses its own library BFD to read your files. GDB knows whether it is reading an *executable*, a *core*, or a *.o* file; however, you can specify the file format with the `set gnutarget` command. Unlike most `target` commands, with `gnutarget` the `target` refers to a program, not a machine.

> *Warning:* To specify a file format with `set gnutarget`, you must know the actual BFD name.

See Section 12.1 [Commands to specify files], page 109.

`show gnutarget`
: Use the `show gnutarget` command to display what file format `gnutarget` is set to read. If you have not set `gnutarget`, GDB will determine the file format for each file automatically, and `show gnutarget` displays '`The current BDF target is "auto"`'.

Here are some common targets (available, or not, depending on the GDB configuration):

`target exec` *program*
: An executable file. '`target exec` *program*' is the same as '`exec-file` *program*'.

`target core` *filename*
: A core dump file. '`target core` *filename*' is the same as '`core-file` *filename*'.

`target remote` *dev*
: Remote serial target in GDB-specific protocol. The argument *dev* specifies what serial device to use for the connection (e.g. '`/dev/ttya`'). supports the `load` command. This is only useful if you have some other way of getting the stub to the target system, and you can put it somewhere in memory where it won't get clobbered by the download.

`target sim`
: Builtin CPU simulator. GDB includes simulators for most architectures. In general,

```
target sim
load
run
```

works; however, you cannot assume that a specific memory map, device drivers, or even basic I/O is available, although some simulators do provide these.

Some configurations may include these targets as well:

`target nrom` *dev*
: NetROM ROM emulator. This target only supports downloading.

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

Many remote targets require you to download the executable's code once you've successfully established a connection.

`load` *filename*
: Depending on what remote debugging facilities are configured into GDB, the `load` command may be available. Where it exists, it is meant to make *filename* (an executable) available for debugging on the remote system—by downloading, or dynamic linking, for example. `load` also records the *filename* symbol table in GDB, like the `add-symbol-file` command.

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

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

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

13.3 Choosing target byte order

Some types of processors, such as the MIPS, PowerPC, and Hitachi SH, offer the ability to run either big-endian or little-endian byte orders. Usually the executable or symbol will include a bit to designate the endian-ness, and you will not need to worry about which to use. However, you may still find it useful to adjust GDB's idea of processor endian-ness manually.

`set endian big`
: Instruct GDB to assume the target is big-endian.

`set endian little`
: Instruct GDB to assume the target is little-endian.

`set endian auto`
: Instruct GDB to use the byte order associated with the executable.

`show endian`
: Display GDB's current idea of the target byte order.

Note that these commands merely adjust interpretation of symbolic data on the host, and that they have absolutely no effect on the target system.

14 Configuration-Specific Information

While nearly all GDB commands are available for all native and cross versions of the debugger, there are some exceptions. This chapter describes things that are only available in certain configurations.

There are three major categories of configurations: native configurations, where the host and target are the same, embedded operating system configurations, which are usually the same for several different processor architectures, and bare embedded processors, which are quite different from each other.

HP WDB 2.0 provides the following features in addition to the standard GDB features:

- Fix and continue debugging
- Debugging memory problems
- Support for debugging 32-bit and 64-bit programs.
- Support for a Terminal User Interface (TUI) in addition to the GDB line-mode interface, enabled with the -tui option.
- Support for automatic loading of debug information from object modules when an application is compiled with the `+objdebug` option. The benefits of this option are only available if you have installed the most recent linker patch.
- Support for debugging code in shared libraries and archive libraries.
- Support for kernel threads, including thread local storage (HP-UX 11.0 only).
- Support for user threads on HP-UX 10.20 and 11.0.
- Support for hardware watchpoints (HP-UX 11.0 only).
- Support for assembly-level debugging (PA-RISC 1.x and PA-RISC 2.0).
- Support for a subset of XDB commands, enabled with the `-xdb` option.
- Support for Fix and Continue debugging.
- Support for memory leak checking and heap profiling.

14.1 HP-UX Dependencies

Several features available in HP WDB 2.0 depend on specific versions of the linker or compilers.

14.1.1 Linker patch required for +objdebug

You must install the latest linker patch in order to generate object modules that enable faster linking and smaller executable file sizes for large applications. Refer to the HP WDB 2.0 release notes and your compiler release notes for more details.

14.1.2 Fix and Continue compiler dependencies

Fix and Continue is supported only on HP-UX 11.x with these compilers:

- HP C/ANSI C A.11.01.20, or later
- HP aC++ A.03.25, or later
- HP Fortran 90 2.4, or later

Refer to the WDB release notes and your compiler release notes for more details.

14.2 HP-UX Targets

On HP-UX systems, GDB has been configured to support debugging of processes running on the PA-RISC architecture. This means that the only possible targets are:

- An executable that has been compiled and linked to run on HP-UX. This includes binaries that have been marked as `SHMEM_MAGIC`.
- A live HP-UX process, either started by GDB (with the `run` command) or started outside of GDB and attached to (with the `attach` command)
- A core file generated by an HP-UX process that previously aborted execution

GDB on HP-UX has not been configured to support remote debugging, or to support programs running on other platforms.

HP WDB 2.0 can only debug programs compiled with HP aC++, HP's ANSI-compatible C++ compiler. You must use HP DDE to debug programs compiled with HP C++, HP's earlier cfront-based C++ compiler.

If you try to debug cfront-compiled programs, HP WDB 2.0 may fail, or produce confusing results. For example, if you try looking looking at a data member, HP WDB 2.0 may generate a message about illegally accessing memory at `0x0`.

You can detect a cfront executable by using the following commands.

On HP PA-RISC 32-bit programs:

`odump -compunit` *executable file name*

or

On HP PA-RISC 64-bit programs:

```
elfdump -dc executable
```

The cfront compiler emits `HPCPLUSPLUS` while the `aCC` compiler emits `ANSI C++` in the compilation directory.

14.3 Fix and Continue Debugging

Fix and Continue allows you to see the result of changes you make to a program you are debugging without having to re-compile and re-link the entire program.

For example, you can edit a function and use the fix command, which automatically re-compiles your code, links it into a shared library, and continues execution of the program, without leaving the debugger.

Fix and Continue allows you to experiment with various ways of fixing problems until you are satisfied with the correction, before you exit the debugger.

The advantages include:

- You do not have to re-compile and re-link the entire program.
- You do not have to reload the program into the debugger.
- You can resume execution of the program from the fix location.
- You can speed up the development cycle.

> *Note:* Fix and Continue is only supported with the most recent versions of HP C and HP aC++.

In command-line mode, you use the `edit` command before invoking the fix command.

The `edit` command has the following syntax:

`edit` *file1 file2*

where

file represents one or more source files for the current executable. If you do not specify a file name, HP WDB 2.0 edits the currently open source file.

The `fix` command has the following syntax:

`fix` *file1 file2*

where

file represents one or more source files for the current executable. If you do not specify a file name, HP WDB 2.0 re-compiles the currently open source file.

When you edit a file with the edit command and save the changes, the original source file contains the changes, even if you do not use the fix command to re-compile your program in the debugger.

14.3.1 Fix and Continue Restrictions

Fix and Continue has the following restrictions and behaviors:

- You cannot re-compile code that has been optimized.
- You can not add, delete or reorder the local variables and parameters in a function currently active on the stack.
- If you fix a routine in a file that contains function pointers, those function pointers become invalid and will likely cause your program to receive a `SIGSEV` error if the pointers are used.
- You can not change the type of a local variable, file static, global variable, parameter of a function.
- You can not add any function calls that increase the size of the parameter area.
- You can not change a local or file static or global variable to be a register variable, and vice-versa.
- You can not add an `alloca()` function to a frame that did not previously use `alloca()`.
- New structure fields can be added at the end of a structure object, not in the middle of a structure.

  New fields are only accessible by the modified files. Old structure fields remain intact, no swapping of them are permitted.
- If the redefined function is in the call stack but not on the top of the call stack, the modified code will not be executed when the execution resumes.

  The modified function will be executed when it is called next time, or a re-run.
- Breakpoints in the original source file are moved to the modified file. Breakpoints in the already instantiated functions on the call stack in the original file are lost.

- If you change the name of a function and there was a breakpoint set to the old function, HP WDB 2.0 does not move the breakpoint to the new function. The old breakpoint is still valid.
- If the number of lines of the modified file is different from that of the original file, the placement of breakpoints may not be correct.
- When the program resumes, the program counter is moved to the beginning of the same line in the modified function. It is possible that the program counter may be at the wrong line.

14.3.2 Using Fix and Continue

When HP WDB 2.0 re-compiles a fixed source file, it uses the same compiler and the same options that were used to create the original executable. If the compiler generates any syntax errors or it encounters any of the restrictions, HP WDB 2.0 does not patch your changes into the executable image being debugged.

After you successfully re-compile your changes, HP WDB 2.0 uses the fixed version of your code when you use any of the execution commands such as `step`, `run`, or `continue`.

When you use the `edit` command, HP WDB 2.0 then monitors any edited source files for additional changes. After you enter the initial `fix` command, HP WDB 2.0 checks for additional saved changes to a source file each time you enter a program execution command. If a saved source file has been changed, HP WDB 2.0 asks you if you want to fix the changed source, allowing you to apply repeated fixes without explicitly entering the `fix` command.

> *Note:* You must rebuild your program after you use the fix command because the changes you make are temporarily patched into the executable image. The changes are lost if you load a different executable and are not reflected in the original executable when you exit the debugger.

The Fix and Continue allows you to make the following changes:

- Changing existing function definitions.
- Disabling, re-enabling, saving and deleting redefinitions
- Adding global and file static variables.
- Adding new structure fields to the end of a structure type object.
- Setting breakpoints in and single-step within redefined code.

14.3.3 Example Fix and Continue Session

This example shows how you can make and test changes to a function without leaving the debugger session.

Here is a short sample C program with an error.

```
int sum (num)   int num;
{
  int j, total = 0;
  for (j = 0; j <= num; j++)
    total += num;
}
```

```
main()
{
   int num = 10;
   printf("The sum from 1 to %d is = %d\n", num, sum(num));
}
```

1. Compile the program.

   ```
   cc sum.c -g -o mysum

   /usr/ccs/bin/ld: (Warning) At least one PA 2.0 object file (sum.o) was detected.
    The linked output may not run on a PA 1.x system.
   ```

2. Run the program.

   ```
   ./mysum

   The sum from 1 to 10 is = 0
   ```

   This result is obviously wrong. We need to debug the program.

3. Run the debugger.

   ```
   gdb mysum

   Copyright 1986 - 2000 Free Software Foundation, Inc.
   Hewlett-Packard Wildebeest 2.0 (based on GDB 4.17-hpwdb-980821)
   Wildebeest is free software, covered by the GNU General Public License,
   and you are welcome to change it and/or distribute copies of it under
   certain  conditions.  Type "show copying" to see the conditions.
   There is absolutely no warranty for Wildebeest.
   Type "show warranty" for details.
   Wildebeest was built for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
   ..
   ```

   If your `TERM` environment variable is not set to hpterm, start the debugger and set the terminal type for editing in HP WDB 2.0 with this command (ksh shell):

   ```
   TERM=hpterm gdb mysum
   ```

4. Notice that the problem might be that there is no `return` for the `num` function. You can correct this without leaving the debugger.
5. Set a break point at main.

   ```
   (gdb) b main
   Breakpoint 1 at 0x23f8: file sum.c, line 11.
   ```

6. Run your program

   ```
   (gdb) run
   Starting program: /tmp/hmc/mysum

   Breakpoint 1, main () at sum.c:11
   11         int num = 10;
   ```

7. With the program stopped at the break point, you can make changes to the source file with the `edit` command.

   Because you are going to edit the current file, you do not need to specify a source file name.

```
(gdb) edit
```

The edit command opens a new terminal session using your environment variable settings for terminal and editor. The debugger automatically loads the source file.

8. Make the necessary changes. In this case, add

```
return total;
```

to the function named `num`.

9. Save the edited source file and exit the editor. This saves the changes in the actual source file for your program.
10. Use the `fix` command to re-compile your program to see the results of your changes.

```
(gdb) fix
Compiling /dev/src/sum.c...
Linking...
Applying code changes to sum.c.
Fix succeeded.
```

The fix command creates a new executable that includes the changes you made to the source file.

The debugger automatically uses the new executable and picks up the debugging session where you stopped before using the `edit` command.

For example, you can continue stepping through your program and you will find the new `return total;` statement in the source view. You can print the value of `total`, and see that the result is 110.

11. When you are finished with your debugging session, you can exit the debugger normally.

```
(gdb) q
The following modules in /dev/src/mysum have been fixed:
/dev/src/sum.c
Remember to remake the program.
```

The debugger message lists the source files that you have changed during your debugging session.

> *Note:* You must rebuild your program after you use the `fix` command because the changes you make are temporarily patched into the executable image. The changes are lost when you exit the debugger or you load a different executable.

14.4 Debugging Memory Problems

You can use HP WDB 2.0 to find leaks, profile heap usage and detect other heap related errors in HP C, HP aC++, and HP Fortran programs written for HP-UX 10.20 or 11.00 (both 32 bit and 64 bit programs are supported).

On HP-UX 11.00, the memory debugging features of HP WDB 2.0 work with both single-threaded and multi-threaded programs that use POSIX threads.

14.4.1 Memory Debugging Restrictions

Programs with these attributes are not supported:

- CMA or DCE threaded programs on 10.20
- CMA or DCE threaded programs on 11.00 (32-bit and 64-bit)
- `EXEC_MAGIC` executables on HP-UX 10.20 or HP-UX 11.00
- Open Graphics Applications
- 64-bit programs that use a version of the `malloc` package other than the one from the standard C library, `libc.sl`
- Programs that link the archive version of the standard C library, `libc.a`, or the core library, `libcl.a`, on both HP-UX 10.20 and HP-UX 11.00

14.4.2 Using Memory Debugging

HP WDB 2.0 provides several commands that help expose memory-related problems.

The commands allow you to:

- Stop at the free of unallocated or deallocated blocks
- Stop when freeing a block if bad writes occurred outside block boundary
- Stop if a specified block address is allocated or deallocated
- Scramble previous memory contents at malloc/free calls
- Report memory leaks

To debug memory problems, use the commands:

`set heap-check leaks [on | off]`
: Controls HP WDB 2.0 memory leak checking.

`show heap-check`
: Displays all current settings for memory checking.

`info leaks`
: Displays a leak report, listing information such as the leaks, size of blocks, number of instances.

`info leaks` *filename*
: Writes leak report output to the specified file.

`info leak` *leaknumber*
: Produces detailed information on the specified leak including the allocation call stack.

`set heap-check block-size` *num-bytes*
: Instructs HP WDB 2.0 to stop the program whenever it tries to allocate a block larger than *num-bytes* in size.

`set heap-check heap-size` *num-size*
: Instructs HP WDB 2.0 to stop the program whenever it tries to increase the program heap by at least *num-bytes*.

`set heap-check frame-count` *num*
: Controls the depth of the call stack collected. Larger values increase run time. Default value is four (4) stack frames.

`set heap-check min-leak-size` *num*
: Collects a stack trace only if the size of the leak exceeds the number of bytes you specify for this value. Larger values improve run-time performance. Default value is zero (0) bytes.

`set heap-check free [on | off]`
: When set to on, forces HP WDB 2.0 to stop your program when it detects a call to `free()` with an improper argument or a call to `realloc()` that does not point to a valid currently allocated heap block.

`set heap-check watch` *address*
: Stops the program whenever a block at a specified address is allocated or deallocated.

`set heap-check bounds [on | off]`
: Allocates extra space at the beginning and end of a heap block during allocation and fills it with a specific pattern. When blocks are freed, HP WDB 2.0 checks to see if these patterns are intact. If they are corrupted, an underflow or overflow must have occurred and HP WDB 2.0 reports the problem. This option increases the program's memory requirements.

`set heap-check scramble [on | off]`
: Scrambles a memory block and overwrites it with a specific pattern any time it is allocated or deallocated. This change to the memory contents increases the chance that erroneous behaviors such as attempting to access space that is freed or depending on initial values of `malloc()` blocks cause the program to fail.

14.4.3 Stop at the free of unallocated or deallocated blocks

HP WDB 2.0 allows you to locate improper calls to `free()` and `realloc()` with the command `set heap-check free [on | off]`.

With this setting on, whenever your program calls `free()` or `realloc()` HP WDB 2.0 inspects the parameters to verify that they are pointing to valid currently allocated heap blocks.

If HP WDB 2.0 detects an erroneous call to `free()`, it stops the program and reports this condition. You can then look at the stack trace to understand where and how the problem occurred.

14.4.4 Stop when freeing a block if bad writes occurred outside block boundary

HP WDB 2.0 allows you to locate problems caused by heap block overflow or underflow with the command `set heap-check bounds [on | off]`. When bounds checking is turned on, HP WDB 2.0 allocates extra space at the beginning and end of a block during allocation and fills it up with a specific pattern. When blocks are freed, HP WDB 2.0 checks to see if these patterns are intact. If they are corrupted, an underflow or overflow must have occurred and HP WDB 2.0 reports the problem.

> *Note:* Turning on bounds checking increases the program's memory requirements because the extra guard bytes must be allocated at the beginning and end of each block.

14.4.5 Stop if a specified block address is allocated or deallocated

To stop a program whenever a block at a specified address is allocated or deallocated use the command:

`set heap-check watch` *address*

You could use this to debug situations such as multiple `free()` calls to the same block.

14.4.6 Scramble previous memory contents at malloc/free calls

HP WDB 2.0 allows you to potentially expose latent memory access defects with the command `set heap-check scramble [on | off]`.

When this setting is turned on, any time a memory block is allocated or deallocated, HP WDB 2.0 scrambles the space and overwrites it with a specific pattern.

This change to the memory contents increases the chance that erroneous behaviors will cause the program to fail. Examples of such behavior include, attempting to access space that is freed or depending on initial values of `malloc()` blocks.

You can now look at the stack trace to understand where and how the problem occurred.

> *Note:* Turning on scrambling will slow down the program slightly, because at every `malloc()` and `free()` call, the space involved must be overwritten.

14.4.7 When to suspect a memory leak

You should suspect a memory leak in your code if you notice your system running out of swap space or running slower and slower, or both.

Applications or non-kernel code (including daemons) that have memory leaks can eventually use up all swap space. You can run `top(1)` to see if your process data space (`SIZE`, `RES`) is growing more than you expect.

If your system is running out of swap space, programs will fail with out of memory (`ENOMEM`) errors or `SIGBUS` signals. In addition, the system might run slower and slower until it comes to a stop; all processes requiring swap to continue running will wait for it indefinitely.

14.4.8 Report memory leaks

HP WDB 2.0 allows you to check for memory leaks in applications written for HP-UX 10.20 and 11.00 using C, ANSI C++, FORTRAN 77 and Fortran 90.

You can use two methods to identify heap-related problems:

- In batch mode to create a list of leaks in your programs
- Interactively by using the `set heap-check leaks [on | off]` command

14.4.9 Detecting memory leaks in batch mode

Enabling batch mode allows you to detect leaks by exercising your program with a test suite. This mode generates a leak report, but no stack traces, when your program run completes. Note that HP WDB 2.0 generates a leak report only if your program terminates with an explicit call to `exit()`.

You can use the batch mode only with non-threaded programs. Only leak checking is available in batch, all other analysis and diagnostics are disabled.

The leak report is sent to a file named `/tmp/gdbrtc.log`. Each run appends its output to the end of the file.

To use this feature, you must do the following:

1. For 32-bit programs, link with `/opt/langtools/lib/librtc.a`. For 64-bit programs link with `/opt/langtools/lib/pa20_64/librtc.a`. The file `librtc.a` must be listed before the C library in the link line.
2. If your program links with either the archive version of the C library, `/usr/lib/libc.a` or the core libraries, `/usr/lib/libcl.a`, you must link with the corresponding shared version instead.
3. Link your application with the core libraries with the `-lcl` linker option.
4. Run your program's test suite.
5. Review the file named `/tmp/gdbrtc.log`, which contains a list of memory leaks.

14.4.10 Configuring memory debugging

Several settings are available to control the amount of detail you can see when debugging memory leaks.

14.4.10.1 Specifying the stack depth

By default, HP WDB 2.0 reports each memory leak with at most four stack frames from the allocating call stack.

The depth of the call stack can be controlled by the command:

`set heap-check frame-count` *num*

Specifying a higher value helps you understand the allocation scenario better. However, the higher the value, the slower your program runs.

Larger stack depth values cause slower performance because HP WDB 2.0 must collect the stack trace for every allocation. The more levels of stack trace you want, the more time it takes to collect the stack trace.

HP WDB 2.0 takes more time because it cannot determine at the time a block is allocated if that block will eventually be leaked by the program.

For many programs, the default value should be appropriate.

14.4.10.2 Specifying minimum leak size

HP WDB 2.0 allows you to specify the minimum leak size for stack trace collection to improve the program's performance.

Stack trace collection slows down your program because it occurs on every allocation call. Therefore, if your program is allocation intensive, HP WDB 2.0 could spend a substantial amount of time collecting stack traces.

You could speed things up by using the command

`set heap-check min-leak-size` *num*

For example, if you use,

```
set heap-check min-leak-size 100
```

HP WDB 2.0 does not collect stack traces for allocations less than 100 bytes. HP WDB still reports leaks smaller than this size, but does not include a stack trace.

14.4.11 Detecting memory leaks interactively

The `set heap-check leaks [on | off]` command controls HP WDB 2.0 memory leak checking.

With leak checking turned on, you can generate a leak report any time the program is stopped inside the debugger, by using the command:

```
info leaks
```

When you use `info leaks`, HP WDB 2.0 scans for memory leaks in your program and produces a leak report, listing information such as the leaks, size of blocks, number of instances. HP WDB 2.0 assigns each of the leaks a unique number.

For additional leak information, use the command `info leak` *leaknumber* to display detailed information on the leak including the allocation call stack.

Use these steps to report leaks:

1. For 64-bit programs, link your application with `/opt/langtools/lib/pa20_64/librtc.sl`.

   > *Note:* If you are using the static or dynamic linker version earlier than B.11.17, you must link your 32-bit program with `/opt/langtools/lib/librtc.sl`.

2. If your program links with either the archive version of the C library, `/usr/lib/libc.a`, or the core libraries, `/usr/lib/libcl.a`, you must link with the corresponding shared version instead.
3. Invoke HP WDB 2.0 with the command-line option `-leaks` or use the command `set heap-check leaks on` after starting the debugger.
4. Any time your application is stopped in the debugger, you can ask for a leak report with the command `info leaks`.

   To see detailed information associated with a leak use the command `info leak` *leaknumber*

14.4.11.1 Example

This example describes checking a program running on HP-UX 11.00 using linker version B.11.17 or later.

1. For a 64-bit program, link your application with `/opt/langtools/lib/pa20_64/librtc.sl`.
2. Link your program with `/usr/lib/libc.sl` instead of `libc.a`.
3. Run the debugger and load your program.

   ```
   > gdb a.out
   ```
4. Turn on leak checking.

   ```
   (gdb) set heap-check leaks on
   ```
5. Set one or more breakpoints in your code where you want to examine cumulative leaks.

   ```
   (gdb) b myfunction
   ```
6. Run your program in the debugger.

   ```
   (gdb) run
   ```
7. Use the `info` command to show list of memory leaks.

   ```
   (gdb) info leaks

   Scanning for memory leaks...done

   2439 bytes leaked in 25 blocks

   No.   Total bytes     Blocks     Address     Function
   0        1234           1       0x40419710   foo()
   1        333            1       0x40410bf8   main()
   2        245            8       0x40410838   strdup()

   [...]
   ```

   The debugger assigns each leak a numeric identifier.
8. To display a stack trace for a specific leak, use the `info leak` command and specify the number from the list associated with a leak.

   ```
   (gdb) info leak 2

   245 bytes leaked in 8 blocks (10.05% of all bytes leaked)
   These range in size from 26 to 36 bytes and are allocated
    in strdup ()
    in link_the_list () at test.c:55
    in main () at test.c:13
    in _start ()
   ```

14.5 Heap Profiling

Seeing the heap profile is useful for identifying how memory is being used by the program. You can use HP WDB 2.0 to profile an application's current heap usage.

`info heap` Displays a heap report, listing information such as the heap allocations, size of blocks, number of instances. Note that the report shows heap usage at the point you use the `info heap` command.

The report does not show allocations that have already been freed. For example, if you make several allocations, free them all, and then you use `info heap`, the result will not show any allocations.

`info heap` *filename*
: Writes heap report output to the specified file.

`info heap` *idnumber*
: Produces detailed information on the specified heap allocation including the allocation call stack.

`set heap-check frame-count` *num*
: Controls the depth of the call stack collected. Larger values increase run time. Default value is four (4) stack frames.

`show heap-check`
: Displays all current settings for memory checking.

Here is an example that shows how to use this feature on HP-UX 11.00 with linker version B.11.17 or later installed.

1. Link with `/opt/langtools/lib/pa20_64/librtc.sl` for PA-64 programs. Note: If your linker version is earlier than B.11.17, and you are debugging a 32-bit program, you must link with `/opt/langtools/lib/librtc.sl`.
2. Turn on profiling with the `set heap-check on` command.

   ```
   (gdb) set heap-check on
   ```
3. Set a breakpoint.

   ```
   (gdb) b myfunction
   ```
4. When the program is stopped at a breakpoint, use the `info heap` command.

   ```
   (gdb) info heap

   Analyzing heap ...done

   41558 bytes allocated in 28 blocks

   No.   Total bytes     Blocks     Address     Function
   0         34567         1       0x40411000   foo()
   1         4096          1       0x7bd63000   bar()
   2         1234          1       0x40419710   baz()
   3         245           8       0x404108b0   boo()

   [...]
   ```

   The display shows the currently allocated heap blocks. Any blocks that have been allocated and already freed, are not listed.

To look at a specific allocation, specify the allocation number with the `info heap` command.

```
(gdb) info heap 1
4096 bytes at 0x7bd63000 (9.86% of all bytes allocated)
  in bar () at test.c:108
```

```
  in main () at test.c:17
  in _start ()
  in $START$ ()
```

When there are multiple blocks allocated from the same call stack, HP WDB 2.0 displays additional information.

```
(gdb) info heap 3
245 bytes in 8 blocks (0.59% of all bytes allocated)
These range in size from 26 to 36 bytes and are allocated
  in boo ()
  in link_the_list () at test.c:55
  in main () at test.c:13
  in _start ()
```

14.6 Specifying object file directories

GDB enables automatic loading of debug information from object modules when an application is compiled with the `+objdebug` option.

GDB uses the full path name to the object module files and searches the same directories for source files.

This behavior transparent, however, you can control over when and how object files are loaded with three commands.

`objectdir` *path*
: Specify a colon (:) separated list of directories in which GDB searches for object files. These directories are added to the beginning of the existing `objectdir` path. If you specify a directory that is already in the `objectdir` path, the specified directory is moved up in the `objectdir` path so that it is searched earlier.

  GDB recognizes two special directory names: `$cdir`, which refers to the compilation directory (if available) and `$cwd`, which tracks GDB's current working directory.

`objectload` *file.c*
: Cause GDB to load the debug information for *file.c* immediately. The default is to load debug information from object modules on demand.

`objectretry` *file.c*
: Force GDB to retry loading an object file if GDB encounters a file error while reading an object module. File errors that might cause this include incorrect permissions, file not found, or if the `objectdir` path changes. By default, GDB does not try to read an object file after an error.

Here are some items to check if the debugger can not find your source files.

1. Make certain the files were compiled with the `-g` switch. Type *`info sources`* to find the list of files that the debugger knows were compiled with `-g`.
2. Make certain that the debugger can find the source file. Type *`show dir`* to find the list of directories the debugger uses to search for source files and type *`set dir`* to change that path.

On HP-UX, the debug information does not contain the full path name to the source file, only the relative pathname that was recorded at compile time. Consequently, you may need several `dir` commands for a complex application with multiple source directories. One way to do this is to place them in a '`.gdbinit`' file placed in the directory used to debug the application.

A sample of the '`.gdbinit`' file might look like the following:

```
dir /home/fred/appx/system
dir /home/fred/appx/display
dir /home/fred/appx/actor
dir /home/fred/appx/actor/sys
     ...
```

Note, When you compile your program with the `+objdebug` option, the debugger may find your source files without using the `dir` command. This happens because the debugger stores the full path name to the object files and searches for source files in the same directories.

14.7 Stopping and starting in shared libraries

On HP-UX, shared libraries are special. Until the library is loaded, GDB does not know the names of symbols. However, GDB gives you two ways to set breakpoints in shared libraries.

- deferred breakpoints
- `catch load` command

14.7.1 Deferred breakpoints

On HP-UX, GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library—unless you are debugging a core file).

When you specify a breakpoint using a name that GDB does not recognize, the debugger warns you with a message that it is setting a deferred breakpoint on the name you specified. If any shared library is loaded with a matching name then GDB sets the breakpoint.

For example, if you type:

'`break foo`'

the debugger does not know if `foo` is a misspelled name or if it is the name of a routine that has not yet been loaded from a shared library. The debugger displays a warning message that it is setting a deferred breakpoint on `foo`. If any shared library is loaded that contains a `foo`, then GDB sets the breakpoint.

If this is not what you want, for example the name was mis-typed, then you can delete the breakpoint.

14.7.2 Using `catch load`

The command '`catch load` <*libname*>' causes the debugger to stop when the particular library is loaded. This gives you a chance to set breakpoints before routines are executed.

14.7.3 Privately mapping shared libraries

In cases where you attach to a running program and you try to set a breakpoint in a shared library, GDB may generated the following message:

```
The shared libraries were not privately mapped; setting a breakpoint
in a shared library will not work until you rerun the program.
```

GDB generates this message because the debugger sets breakpoints by replacing an instruction with a `BREAK` instruction. The debugger can not set a breakpoint in a shared library because doing so can affect other processes on the system in addition to the process being debugged.

To set the breakpoint you must kill the program and then re-run it so that the dynamic linker will map a copy of the shared library. There are two ways to run the program:

- Re-run the program under GDB to have the debugger tell `dld` to map all shared libraries private, enabling breakpoint debugging.
- Use the following command on an executable:

  '`/opt/langtools/bin/pxdb -s on` *executable-name*'

  The `pxdb -s on` command marks the executable so that `dld` maps shared libraries private when the program starts up.

14.8 Using shared library as main program

My main program is in a shared library. I try to load it as follows:

```
(gdb) symbol-file main.sl
Load new symbol table from "main.sl"? (y or n) y
Reading symbols from main.sl
done.
```

Things don't appear to work.

This command is not the correct thing to do. This command assumes that '`main.sl`' is loaded at its link time address. This is not not true for shared libraries.

Do not use `symbol-file` with shared libraries.

Instead, what you should do is to use the deferred breakpoint feature to set breakpoints on any functions necessary before the program starts running.

```
(gdb) b main
Breakpoint 1 (deferred) at "main" ("main" was not found).
Breakpoint deferred until a shared library containing "main" is loaded.
(gdb) r
```

Once the program has started running, it will hit the breakpoint. In addition, the debugger will then already know about the sources for `main`, since it gets this information when the shared library is loaded.

14.9 Getting information from a non-debug executable

You can get some information about the arguments passed to the functions displayed in the stack trace in a non-debug, optimized executable.

GDB has no debug information, it does not know where the arguments are located or even the type of the arguments. There is no way for GDB to infer this in an optimized, non-debug executable.

However, for integer arguments you can find the first four parameters for the top-of-stack frame by looking at the registers. The first parameter will be in `$r26`, the second in `$r25` and so on.

14.10 Parallel Processing

HP WDB 2.0 supports `pthread` parallelism.

However, HP WDB 2.0 does not support compiler generated parallelism, e.g. with directives.

14.11 Support for ddd

GDB works with `ddd` the free GDB GUI shell available at `http://mumm.ibr.cs.tu-bs.de/`.

While this is not formally supported by Hewlett-Packard, these two do work together. Note however if you have `ddd` issues, you'll need to report them to the `ddd` support channel.

15 The HP-UX Terminal User Interface

By default, GDB runs in line mode. For users who prefer an interface similar (though not identical) to that of the XDB debugger, HP provides a terminal user interface (TUI), which appears when you invoke the `gdb` command with the `-tui` option.

Use the `-xdb` option to allow the use of a number of XDB commands. See the Chapter 16 [XDB to WDB Transition Guide], page 151.

This guide contains the following topics:

- Starting the TUI
- Screen Layouts
- Cycling Through the Windows
- Changing Window Focus
- Scrolling Windows
- Changing the Register Display
- Changing Window Size
- Refreshing and Updating the Screen

15.1 Starting the TUI

Invoke the debugger using a command like the following:

gdb -xdb -tui a.out

These examples use the default terminal screen size of 24 by 80 characters. The terminal screen looks something like this.

Figure 1

```
   |----------------------------------------------------------------------|
   |30      {                                                             |
   |31          /* Try two test cases. */                                 |
   |32          print_average (my_list, first, last);                     |
   |33          print_average (my_list, first, last - 3);                 |
   |34      }                                                             |
   |35                                                                    |
   |36                                                                    |
   |37                                                                    |
   |38                                                                    |
   |39                                                                    |
   |40                                                                    |
   |41                                                                    |
   |42                                                                    |
   |----------------------------------------------------------------------|
 File: average.c    Procedure: ??    Line: ??      pc: ??
Wildebeest is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under certain
conditions.  Type "show copying" to see the conditions.  There is
absolutely no warranty for Wildebeest.  Type "show warranty" for details.
---Type <return> to continue, or q <return> to quit---
Wildebeest was built for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
..
(gdb)
```

The terminal is divided into two windows: a source window at the top and a command window at the bottom. In the middle is a *locator bar* that shows the current file, procedure, line, and program counter (PC) address, when they are known to the debugger.

When you set a breakpoint on the main program by issuing the command

b main

an asterisk (*) appears opposite the first executable line of the program. When you execute the program up to the first breakpoint by issuing the command

run

a right angle bracket (>) points to the current location. So after you issue those commands, the screen looks something like this:

Figure 2

```
   |--------------------------------------------------------------------|
   |27      }                                                           |
   |28                                                                  |
   |29      int main(void)                                              |
   |30      {                                                           |
   |31          /* Try two test cases. */                               |
 *>|32          print_average (my_list, first, last);                   |
   |33          print_average (my_list, first, last - 3);               |
   |34      }                                                           |
   |35                                                                  |
   |36                                                                  |
   |37                                                                  |
   |38                                                                  |
   |39                                                                  |
   |--------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
..
(gdb) b main
Breakpoint 1 at 0x3524: file average.c, line 32.
(gdb) run
Starting program: /tmp_mnt/home/roz/roz/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb)
```

15.2 Automatically running a program at startup

HP WDB 2.0 does not start running the target executable at startup as do '`xdb`' and HP DDE. This makes it easy to set break points before the target program's main function.

To make HP WDB 2.0 automatically start running the target program add these lines to your startup file, '`.gdbinit`':

```
break main
start
```

15.3 Screen Layouts

The TUI supports four windows within the terminal screen, in various combinations:

- Command
- Source
- Disassembly
- Register

The command window is always present. The possible screen configurations of the other windows are:

- Source
- Disassembly
- Source/Disassembly
- Disassembly/Register
- Source/Register

The `layout` command (abbreviated `la`) allows you to change from one window configuration to another.

> **Note:** You can abbreviate any command to its shortest unambiguous form.

15.3.1 Source Window

The Source window, Figure 1, appears by default when you invoke the debugger. You can also make it appear by issuing the command

```
la src
```

15.3.2 Disassembly Window

The Disassembly window appears when you issue the command

```
la asm
```

The screen looks like this:

Figure 3

```
  |--------------------------------------------------------------------|
  |;;;     print_average (my_list, first, last);                       |
*>|0x3524 <main+8> addil L'-0x800,%dp,%r1                              |
  |0x3528 <main+12>       ldo 0x730(%r1),%r26                          |
  |0x352c <main+16>       ldi 9,%r24                                   |
  |0x3530 <main+20>       ldi 0,%r25                                   |
  |0x3534 <main+24>       ldil L'0x3000,%r31                           |
  |0x3538 <main+28>       be,l 0x498(%sr4,%r31)                        |
  |0x353c <main+32>       copy %r31,%rp                                |
  |;;;     print_average (my_list, first, last - 3);                   |
  |0x3540 <main+36>       addil L'-0x800,%dp,%r1                       |
  |0x3544 <main+40>       ldo 0x730(%r1),%r26                          |
  |0x3548 <main+44>       ldi 6,%r24                                   |
  |0x354c <main+48>       ldi 0,%r25                                   |
  |--------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) b main
Breakpoint 1 at 0x3524: file average.c, line 32.
(gdb) r
Starting program: /tmp_mnt/home/roz/roz/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb)
```

15.3.3 Source/Disassembly Window

The Source/Disassembly window appears when you issue the command

```
la split
```

You can also reach this window from the Source window with the XDB command

```
td
```

The screen looks like this:

Figure 4

```
   :......................................................................:
 *>:32          print_average (my_list, first, last);                     :
   :33          print_average (my_list, first, last - 3);                 :
   :34      }                                                             :
   :35                                                                    :
   :36                                                                    :
   :37                                                                    :
   :......................................................................:
   |;;;     print_average (my_list, first, last);                         |
 *>|0x3524 <main+8> addil L'-0x800,%dp,%r1                                |
   |0x3528 <main+12>        ldo 0x730(%r1),%r26                           |
   |0x352c <main+16>        ldi 9,%r24                                    |
   |0x3530 <main+20>        ldi 0,%r25                                    |
   |0x3534 <main+24>        ldil L'0x3000,%r31                            |
   |----------------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
Breakpoint 1 at 0x3524: file average.c, line 32.
(gdb) r
Starting program: /tmp_mnt/home/roz/roz/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb)
```

15.3.4 Disassembly/Register Window

The Disassembly/Register window appears when you issue the command

```
la regs
```

when the current window is the Source/Disassembly window. By default, the debugger displays the general registers.

The screen looks like this:

Figure 5

```
:...................................................................:
:flags 29000041         r1 51a800               rp 7f6ce597         :
:r3 7f7f0000            r4 1                    r5 7f7f06f4         :
:r6 7f7f06fc            r7 7f7f0800             r8 7f7f0800         :
:r9 40006b10            r10 0                   r11 40004b78        :
:r12 1                  r13 0                   r14 0               :
:r15 0                  r16 40003fb8            r17 4               :
:...................................................................:
   |;;;     print_average (my_list, first, last);                   |
 *>|0x3524 <main+8> addil L'-0x800,%dp,%r1                          |
   |0x3528 <main+12>       ldo 0x730(%r1),%r26                      |
   |0x352c <main+16>       ldi 9,%r24                               |
   |0x3530 <main+20>       ldi 0,%r25                               |
   |0x3534 <main+24>       ldil L'0x3000,%r31                       |
   |----------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) r
Starting program: /tmp_mnt/home/roz/roz/work/wdb/a.out

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb)
```

15.3.5 Source/Register Window

The Source/Register window appears when you issue the command

```
la regs
```

when the current window is the Source window.

The screen looks like this:

Figure 6

```
:...................................................................:
:flags 29000041         r1 51a800              rp 7f6ce597          :
:r3 7f7f0000            r4 1                   r5 7f7f06f4          :
:r6 7f7f06fc            r7 7f7f0800            r8 7f7f0800          :
:r9 40006b10            r10 0                  r11 40004b78         :
:r12 1                  r13 0                  r14 0                :
:r15 0                  r16 40003fb8           r17 4                :
:...................................................................:
 *>|32         print_average (my_list, first, last);                |
   |33         print_average (my_list, first, last - 3);            |
   |34     }                                                        |
   |35                                                              |
   |36                                                              |
   |37                                                              |
   |----------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb)
```

15.4 Cycling Through the Windows

Use the commands

```
la next
```

and

```
la prev
```

to move from one screen configuration to another without specifying a window name. If you specify `la next` repeatedly, the order the debugger uses is

- Source (`src`)
- Disassembly (`asm`)
- Source/Disassembly (`split`)
- Source/Register
- Disassembly/Register

If you invoked the `gdb` command with the `-xdb` option as well as the `-tui` option, you can also use the following commands:

`td` Toggle between Source and Disassembly/Register windows.

`ts` Toggle split-screen mode.

15.5 Changing Window Focus

The command window always has keyboard focus, so that you can enter debugger commands. If there is only one other window (Source or Disassembly), the other window has the *logical* focus, so that you can scroll within that window by using the arrow keys or the ⟨Page Up⟩ and ⟨Page Down⟩ keys (on some keyboards these are ⟨Prev⟩ and ⟨Next⟩).

Note: In the command window, the scrolling behavior only works for an '`hpterm`' and not for an '`xterm`' or '`dtterm`'.

If you are in split-screen mode, you may want to change the logical focus of the window. To do so, use the command

`focus {`*win_name*` | prev | next}`

where *win_name* can be `src`, `asm`, `regs`, or `cmd`.

Remember, if you change the focus to a window other than the command window, you need to use `focus cmd` to switch back to the command window to enter or scroll through commands.

For example, with the sequence of commands just issued, you are in split-screen mode with the focus in the Source window.

The window with logical focus has a border constructed from `"|"` and `"-"`.

A window that does not have logical focus has a border constructed from `":"`vand `"."`:

Figure 7

```
:..................................................................:
:flags 29000041         r1 51a800              rp 7f6ce597          :
:r3 7f7f0000            r4 1                   r5 7f7f06f4          :
:r6 7f7f06fc            r7 7f7f0800            r8 7f7f0800          :
:r9 40006b10            r10 0                  r11 40004b78         :
:r12 1                  r13 0                  r14 0                :
:r15 0                  r16 40003fb8           r17 4                :
:..................................................................:
 *>|32          print_average (my_list, first, last);               |
   |33          print_average (my_list, first, last - 3);           |
   |34      }                                                       |
   |35                                                              |
   |36                                                              |
   |37                                                              |
   |----------------------------------------------------------------|
 File: average.c    Procedure: main    Line: 32      pc: 0x3524

Breakpoint 1, main () at average.c:32
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb)
```

By default, the Source window will scroll. To change the focus so that you can scroll in the Register window, use the `focus` command (abbreviated `foc` or `fs`):

```
fs regs
```

or

```
foc next
```

If you then use the ⟨Page Down⟩ key to scroll in the Register window, the screen looks like this:

Figure 8

```
|---------------------------------------------------------------------------|
|flags 29000041         r1 51a800               rp 7f6ce597                 |
|r3 7f7f0000            r4 1                    r5 7f7f06f4                 |
|r6 7f7f06fc            r7 7f7f0800             r8 7f7f0800                 |
|r9 40006b10            r10 0                   r11 40004b78                |
|r12 1                  r13 0                   r14 0                       |
|r15 0                  r16 40003fb8            r17 4                       |
|---------------------------------------------------------------------------|
 *>:32          print_average (my_list, first, last);                       :
   :33          print_average (my_list, first, last - 3);                   :
   :34      }                                                               :
   :35                                                                      :
   :36                                                                      :
   :37                                                                      :
   :.......................................................................:
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) la asm
(gdb) la split
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb) foc next
Focus set to REGS window.
(gdb)
```

15.6 Scrolling Windows

To scroll a window, you can use the arrow keys or the ⟨Page Up⟩ and ⟨Page Down⟩ keys (on some keyboards these are ⟨Prev⟩ and ⟨Next⟩). You can also use the following commands:

`{+ | -} [`*num_lines*`] [`*win_name*`]`

Vertically scroll the window forward (`+`) or backward (`-`). `+` or `-` with no arguments scrolls the window forward or backward one page. Use *num_lines* to specify how many lines to scroll the window. Use *win_name* to specify a window other than the one with logical focus.

`{< | >}[`*num_char*`] [`*win_name*`]`

Horizontally scroll the window left (`<`) or right (`>`) the specified number of characters. If you do not specify *num_char*, the window is scrolled one character.

Note that a space is required between the `+`, `-`, `<`, or `>` and the number.

To scroll the command window, use the scroll bars on the terminal window.

15.7 Changing the Register Display

To look at the floating-point or special registers instead of the general registers, and then to return to the general registers, you can use the following XDB commands:

`fr`
`display $fregs`
: Display the floating-point registers.

`sr`
`display $sregs`
: Display the special registers.

`gr`
`display $gregs`
: Display the general registers.

For example, if you use the `fr` command, the screen looks like this:

Figure 9

```
|--------------------------------------------------------------------------|
|flags 29000041          r1 51a800              rp 7f6ce597                |
|r3 7f7f0000             r4 1                   r5 7f7f06f4                |
|r6 7f7f06fc             r7 7f7f0800            r8 7f7f0800                |
|r9 40006b10             r10 0                  r11 40004b78               |
|r12 1                   r13 0                  r14 0                      |
|r15 0                   r16 40003fb8           r17 4                      |
   :.......................................................................:
   :30      {                                                              :
   :31          /* Try two test cases. */                                  :
 *>:32          print_average (my_list, first, last);                      :
   :33          print_average (my_list, first, last - 3);                  :
   :34      }                                                              :
   :35                                                                     :
   :.......................................................................:
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb) foc next
Focus set to REGS window.
(gdb) fr
#0  main () at average.c:32
(gdb)
```

The default floating-point register display is single-precision. To change the register display to double-precision and then back again, use the XDB toggle float command:

```
toggle $fregs
```

The screen looks like this:

Figure 10

```
|--------------------------------------------------------------------------|
|fpsr 0                         fpe1 0                                     |
|fpe2 0                         fpe3 0                                     |
|fpe4 0                         fpe5 0                                     |
|fpe6 0                         fpe7 0                                     |
|fr4    0                       fr4R    0                                  |
|fr5    1.0000000000000011      fr5R    7.00649232e-45                     |
|--------------------------------------------------------------------------|
 *>:32         print_average (my_list, first, last);                       :
   :33         print_average (my_list, first, last - 3);                   :
   :34     }                                                               :
   :35                                                                     :
   :36                                                                     :
   :37                                                                     :
   :.......................................................................:
 File: average.c    Procedure: main    Line: 32      pc: 0x3524
(gdb) la regs
(gdb) la src
(gdb) la regs
(gdb) foc next
Focus set to REGS window.
(gdb) fr
(gdb) tf
(gdb)
```

15.8 Changing Window Size

To specify a new height for a window or to increase or decrease the current height, use the `winheight` command (abbreviated `winh` or `wh`).

The syntax is:

`winheight [`*win_name*`] [+ | -]` *num_lines*

If you omit *win_name*, the window with logical focus is resized. When you increase the height of a window, the height of the command window is decreased by the same amount, and vice versa. The height of any other windows remains unchanged.

For example, the command

```
wh src +3
```

increases the size of the source window, and decreases the size of the command window, by 3 lines.

To find out the current sizes of all windows, use the `info win` command. For example, if you have a split-screen layout, the command output might be as follows:

```
(gdb) i win
        SRC     (8 lines)
        REGS    (8 lines)
        CMD     (8 lines)
```

If you use the mouse or window menus to resize the terminal window during a debugging session, the screen remains the same size it was when you started. To change the screen size, you must exit the debugger and restart it.

15.9 Refreshing and Updating the Screen

If the screen display is disrupted for some reason, use the `refresh` command (`ref`) to restore the windows to their previous state:

```
ref
```

If you use stack-navigation commands such as `up`, `down`, and `frame` to change your source location, and you wish to return the display to the current point of execution, use the `update` command (`upd`):

```
upd
```

16 XDB to WDB Transition Guide

This is a transition aid designed for XDB users who are learning HP WDB 2.0, an HP-supported version of the industry-standard GDB debugger. Select one of these lists for a table that shows HP WDB 2.0 equivalents for many common XDB commands and other features.

Invoke HP WDB 2.0 with the command `gdb -tui` to obtain a terminal user interface (TUI) similar to that provided by XDB. Commands marked "(with `-tui`)" are valid when you use the `-tui` option.

Invoke HP WDB 2.0 with the command `gdb -xdb` to turn on XDB compatibility mode, which allows you to use many XDB commands as synonyms for GDB commands. Commands marked "(with `-xdb`)" are valid when you use the `-xdb` option.

You may use both `-xdb` and `-tui` at the same time. Some commands are valid only when you use both options.

For a tutorial introduction to HP WDB 2.0, see Getting Started with HP WDB 2.0 (`GDBtutorial.html`)

16.1 By-Function Lists of XDB Commands and HP WDB Equivalents

- Section 16.2.5 [Global Breakpoint Commands], page 159.
- Section 16.2.6 [Assertion Control Commands], page 160.
- Section 16.2.7 [Record and Playback Commands], page 160.
- Section 16.2.8 [Macro Facility Commands], page 160.
- Section 16.2.9 [Signal Control Commands], page 161.
- Section 16.2.10 [Miscellaneous Commands], page 161.

16.1.1 Invocation Commands

By default, HP WDB runs in line mode. To run it with a terminal user interface similar to that of XDB, use the `-tui` option.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`xdb` *program*	`gdb -xdb` *program*, `gdb -xdb -tui` *program*	Debug program
`xdb` *program corefile*	`gdb -xdb` *program* `-c` *corefile*	Debug core file
`xdb -d` *dir*	`gdb -xdb -d` *dir*	Specify alternate directory to search for source files
`xdb -P` *pid program*	`gdb -xdb` *program pid*	Attach to running program at invocation
`xdb -i`	(after starting) `run <` *file*	Specify input to target program
`xdb -o`	(after starting) `run >` *file*	Specify output from target program

16.1.2 Window Mode Commands

These commands are all TUI mode commands and/or XDB compatibility mode commands. They are available when you invoke HP WDB 2.0 by using the `-tui` or `-xdb` or both options.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`{+ | -}r`	`{+ | -}r` (with `-xdb -tui`), `{+ | -} data` (with `-tui`)	Scroll floating-point registers forward or backward (`src`, `cmd`, and `asm` are also valid window names)
`fr`	`fr` (with `-xdb -tui`), `display $fregs` (with `-tui`)	Display floating-point registers
`gr`	`gr` (with `-xdb -tui`), `display $regs` (with `-tui`)	Display general registers
`sr`	`sr` (with `-xdb -tui`), `display $sregs` (with `-tui`)	Display special registers

`td`	`td` (with `-xdb -tui`)	Toggle disassembly mode
`tf`	`tf` (with `-xdb -tui`), `toggle $fregs` (with `-tui`)	Toggle float register display precision
`ts`	`ts` (with `-xdb -tui`)	Toggle split-screen mode
`u`	`u` (with `-xdb -tui`), `update` (with `-tui`)	Update screen to current execution point
`U`	`U` (with `-xdb -tui`), `refresh` (with `-tui`)	Refresh all windows
`w` *number*	`w` *number* (with `-xdb -tui`), `winheight src` *number* (with `-tui`)	Set size of source window

16.1.3 File Viewing Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
`{+ | -}`[*number*]	`{+ | -}`[*number*] (with `-tui`; note that a space is required between `+` or `-` and the *number*)	Move view location forward or backward in source file *number* lines
`/`[*string*]	`/`[*string*] (with `-xdb`), `search` *regexp*, `forw` *regexp*	Search source forward for [last] *string*
`?`[*string*]	`?`[*string*] (with `-xdb`), `rev` *regexp*	Search source backward for [last] *string*
`D "`*dir*`"`	`D "`*dir*`"` (with `-xdb`), `dir` *pathname*	Add a directory search path for source files
`L`	`L` (with `-xdb`)	Show current viewing location or current point of execution
`ld`	`ld` (with `-xdb`), `show directories`	List source directory search path (list all directories)
`lf`	`lf` (with `-xdb`), `info sources`	List all source files
`lf` [*string*]	No equivalent	List matching files
`n`	`fo` or `rev`	Repeat previous search
`N`	`fo` or `rev`	Repeat previous search in opposite direction
`v`	`v` (with `-xdb`), `list`	Show one source window forward from current
`v` *location*	`v` *location* (with `-xdb`), `list` *location*	View source at *location* in source window

`va` *address*	`va` *address* (with `-xdb`), `disas` *address*	View *address* in disassembly window
`va` *label*	`va` *label* (with `-xdb`), `disas` *label*	View *label* in disassembly window (*label* is a location)
`va` *label* `+` *offset*	`va` *label* `+` *offset* (with `-xdb`), `disas` *label* `+` *offset*	View *label* `+` *offset* in disassembly window

16.1.4 Source Directory Mapping Commands

Use the `D` or `dir` command to add new directories to be searched for source files. See See Section 16.1.3 [XDB-fil], page 153.

GDB does not provide a source directory mapping capability and therefore does not have any equivalent of the `apm`, `dpm`, and `lpm` commands.

16.1.5 Data Viewing and Modification Commands

There are many `info` commands in addition to those shown here. Use `help info` to get a list.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`l`	`l` (with `-xdb`), `info args` followed by `info locals`	List all parameters and locals of current procedure
`lc` [*string*]	`lc` [*string*] (with `-xdb`), `info common` *string*	List all (or matching) commons
`lg` [*string*]	`lg` [*string*] (with `-xdb`), `info variables` [*string*]	List all (or matching) globals
`ll` [*string*]	`info functions` [*string*], `info variables` [*string*], `maint print msymbols` *file*	List the contents of the linker symbol table
`lm`	`show user`	List all string macros
`lm` *string*	`show user` *string*	List matching string macros
`lo` [[*class*]`::`][*string*]	`info func` [[*class*]`::`][*string*]	List all (or matching) overloaded functions
`lp`	`info functions`	Show current scope, list program blocks, list names (symbols)
`lp` [[*class*]`::`]*string*	`info func` [[*class*]`::`]*string* `info addr` [[*class*]`::`]*string*	List all (or matching) procedures
`lr`	`lr` (with `-xdb`), `info all-reg`	List all registers
`lr` *string*	`lr` *string* (with `-xdb`), `info reg` *string*	List matching registers
`ls` [*string*]	No equivalent	List all (or matching) special variables

`mm`	`info sharedlibrary`	Show memory map of all loaded shared libraries
`mm` *string*	No equivalent	Show memory map of matching loaded shared libraries
`p` *expr*[*format*	`p`[`/`*format expr* [Note: The *count* and *size* portions of formats are not allowed in the `p` (`print`) command. They are allowed in the `x` command (examine memory).]	Print value using the specified format
`p` *expr*`?`*format*	`p/`*format* `&`*expr*	Print address using specified format
`p` *class*`::`	No equivalent	Print static members of *class*
`p $lang`	`show language`	Inquire what language is used
`p {+ | -}`[*format*	Use `x/`*format* command to obtain initial value, then use `x` with no argument to obtain value of next memory location. To obtain value of previous memory location, use `"x $_ - 1"`.	Print value of next/previous memory location using *format*
`pq` *expr*	`set` *expr*, `set var` *expr*	Evaluate using the specified format
`pq` *expr*`?`*format*	No equivalent	Determine address using specified format
`pq` *class*`::`	No equivalent	Evaluate static members of *class*
`pq {+ | -}`[*format*	No equivalent	Evaluate next/previous memory location using *format*

16.1.6 Stack Viewing Commands

The GDB concept of the top and bottom of the stack is the opposite of `XDB`'s, so `XDB`'s `up` is GDB's `down`.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`down`	`up`	View procedure one level nearer outermost frame of stack (higher number)
`down` *number*	`up` *number*	View procedure *number* levels nearer outermost frame of stack
`t` [*depth*]	`t` [*depth*] (with `-xdb`), `bt` [*depth*]	Print stack trace to *depth*
`T` [*depth*]	`T` [*depth*] (with `-xdb`), `bt full` [*depth*]	Print stack trace and show local vars

`top`	`frame 0`	View procedure at innermost frame of stack
`up`	`down`	View procedure one level nearer innermost frame of stack (lower number)
`up` *number*	`down` *number*	View procedure *number* levels nearer innermost frame of stack
`V` [*depth*]	`V` [*depth*] (with `-xdb`), `frame` [*depth*]	Display text for current active procedure or at specified *depth* on stack

16.1.7 Status Viewing Command

Type `show` with no arguments to get a list of current debugger settings.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`I`	`info` (many kinds), `show` (many kinds)	Display state of debugger and program

16.1.8 Job Control Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
`c`	`c`, `continue`	Continue from breakpoint, ignoring any pending signal
`c` *location*	`until` *location*	Continue from breakpoint, ignoring any pending signal, set temporary breakpoint at *location*
`C`	`c`, `continue`	Continue, allowing any pending signal
`C` [*location*]	`until` *location*	Continue, allowing any pending signal, set temporary breakpoint at *location*
`g` *line*	`g` *line* (with `-xdb`), `go` *line*, `tb` *line* followed by `jump` *line*	Go to *line* in current procedure
`g #`*label*	No equivalent	Go to *label* in current procedure
`g {+ | -}`*lines*	`g {+ | -}`*lines* (with `-xdb`), `go {+ | -}`*lines*, `tb {+ | -}`*lines* followed by `jump {+ | -}`*lines*	Go forward or back given # lines
`g {+ | -}`	`g {+ | -}` (with `-xdb`), `go {+ | -}1`, `tb {+ | -}1` followed by `jump {+ | -}1`	Go forward or back 1 line
`k`	`k`	Detach and terminate target

`r` [*arguments*]	`r` [*arguments*]	Run with last arguments [or with new arguments]
`R`	`R` (with `-xdb`), `r`	Rerun with no arguments
`s`	`s`, `si`	Single step (into procedures) (`si`: step by instruction)
`s` *number*	`s` *number*, `si` *number*	Single step *number* steps (into procedures) (`si`: step by instruction)
`S`	`S` (with `-xdb`), `n`, `ni`	Step over (`ni`: step over by instruction)
`S` *number*	`S` *number* (with `-xdb`), `n` *number*, `ni`*number*	Step over by *number* statements or instructions (`ni`: step over by instruction)

16.2 Overall Breakpoint Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
`lb`	`lb` (with `-xdb`), `i b`	List breakpoints
`tb`	No equivalent	Toggle overall breakpoint state

16.2.1 Auxiliary Breakpoint Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
"*any_string*"	p "*any_string*"	Print *any_string*
`if` *expr* {*cmds*} [{*cmds*}]	`if` *expr* *cmds* [`else` *cmds*] `end`	Conditionally execute *cmds*
`Q`	`Q` (with `-xdb`), `silent` (must be first command in a commands list)	Quiet breakpoints

16.2.2 Breakpoint Creation Commands

The GDB equivalent of the `count` and `cmds` arguments is to use the `commands`*bnum* command to set an ignore count and/or to specify commands to be executed for that breakpoint.

For C++ programs, you can use the regular-expression breakpoint command `rbreak` to set breakpoints on all the member functions of a class or on overloaded functions outside a class.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`b` *loc*	`b` *loc*	Set a breakpoint at the specified location

`b`	`b`	Set a breakpoint at the current line
`ba` *address*	`ba` *address* (with `-xdb`), `b *`*address*	Set breakpoint at a code address
`bb` [*depth*]	No equivalent (use `b` *proc*)	Set breakpoint at procedure beginning
`bi` *expr*`.`*proc*	`b` *class*`::`*proc* `cond` *bnum* `(this ==` *expr*`)`	Set an instance breakpoint at the first executable line of *expr*`.`*proc*
`bi -c` *expr*	No equivalent	Set an instance breakpoint at first executable line of all non-static member functions of the instance's class (no base classes)
`bi -C` *expr*	No equivalent	Set an instance breakpoint at first executable line of all non-static member functions of the instance's class (base classes included)
`bpc -c` *class*	`rb ^`*class*`::*`	Set a class breakpoint at first executable line of all member functions of the instance's class (no base classes)
`bpc -C` *class*	Use `rb ^`*class*`::*` for base classes also	Set a class breakpoint at first executable line of all member functions of the class (base classes included)
`bpo` *proc*	`rb` *proc*	Set breakpoints on overloaded functions outside a class
`bpo` *class*`::`*proc*	`b` *class*`::`*proc*	Set breakpoints on overloaded functions in a class
`bt` [*depth*]	No equivalent	Set trace breakpoint at procedure at specified *depth* on program stack
`bt` *proc*	`b` *proc* `commands` *bnum* `finish c end`	Set trace breakpoint at *proc*
`bu` [*depth*]	`bu` [*depth*] (with `-xdb`). The `finish` command is equivalent to the sequence `bu`, `c`, `db` (to continue out of the current routine).	Set up-level breakpoint
`bx` [*depth*]	`bx` [*depth*] (with `-xdb`)	Set a breakpoint at procedure exit

16.2.3 Breakpoint Status Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
`ab` *number*	`enable` *number*	Activate suspended breakpoint of the given *number*
`ab *`	`enable`	Activate all suspended breakpoints
`ab @`*shared-library*	No equivalent	Activate suspended breakpoints in named shared library
`bc` *number expr*	`bc` *number expr* (with `-xdb`), `ignore`*number expr* (within a commands list)	Set a breakpoint count
`db`	`clear`	Delete breakpoint at current line
`db` *number*	`delete` *number*	Delete breakpoint of the given *number*
`db *`	`delete`	Delete all breakpoints
`sb` *number*	`disable` *number*	Suspend breakpoint of the given *number*
`sb *`	`disable`	Suspend all breakpoints
`sb @`*shared-library*	No equivalent	Suspend breakpoints in named shared library

16.2.4 All-Procedures Breakpoint Commands

GDB does not provide the ability to set breakpoints on all procedures with a single command. Therefore, it does not have any equivalent of the following commands:

```
bp
bpt
bpx
dp
Dpt
Dpx
```

16.2.5 Global Breakpoint Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
`abc` *cmds*	No exact equivalent, but `display` *expr* is equivalent to `abc print` *expr*	Set or delete *cmds* to execute at every stop
`dbc`	`undisplay`	Stop displaying values at each stop

16.2.6 Assertion Control Commands

GDB does not provide the ability to trace by instruction. Watchpoints, however, provide similar functionality to `xdb` assertions.

For example, watchpoints can be:

- Enabled (corresponds to `aa`)
- Disabled (corresponds to `da`)
- Listed (corresponds to `info watch`)
- Added (corresponds to `x`)

HP WDB 2.0 does not have explicit equivalents for the following commands:

```
a
aa
da
la
sa
ta
x
```

16.2.7 Record and Playback Commands

Use the `source` command to read commands from a file. GDB does not provide a recording capability like XDB's, but you can use the `set history save` command to record all GDB commands in the file `./.gdb_history` (similar to the $HOME/.xdbhist file). The history file is not saved until the end of your debugging session.

To change the name of the history file, use `set history filename`.

To stop recording, use `set history save off`.

To display the current history status, use `show history`.

For an equivalent of the XDB record-all facility, pipe the output of the `gdb` command to the *tee*(1) command. For example:

```
gdb a.out | tee mylogfile
```

This solution works with the default line-mode user interface, not with the terminal user interface.

16.2.8 Macro Facility Commands

Use the `show user` or `help user-defined` command to obtain a list of all user-defined commands.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`def` *name replacement-text*	`def` *name* [GDB prompts for commands]	Define a user-defined command
`tm`	No equivalent	Toggle the macro substitution mechanism

`undef` *name*	`def` *name* [follow with empty command list]	Remove the macro definition for *name*
`undef *`	No equivalent	Remove all macro definitions

16.2.9 Signal Control Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
`lz`	`lz` (with `-xdb`), `info signals`	List signal handling
`z` *number* `s`	`z` *number* `s` (with `-xdb`), `handle` *number*`stop`, `handle` *number* `nostop`	Toggle stop flag for signal *number*
`z` *number* `i`	`z` *number* `i` (with `-xdb`), `handle` *number*`nopass`, `handle` *number* `pass`	Toggle ignore flag for signal *number*
`z` *number* `r`	`z` *number* `r` (with `-xdb`), `handle` *number* `print`, `handle` *number* `noprint`	Toggle report flag for signal *number*
`z` *number* `Q`	`z` *number* `Q` (with `-xdb`), `handle` *number* `noprint`	Do not print the new state of the signal

16.2.10 Miscellaneous Commands

XDB Command	HP WDB 2.0 Equivalent	Meaning
Return	*Return*	Repeat previous command
`~`	*Return*	Repeat previous command
`;`	No equivalent (one command per line in command list)	Separate commands in command list
`!` *cmd_line*	`!` *cmd_line* (with `-xdb`), `she` *cmd_line*	Invoke a shell
`{`*cmd_list*`}`	`commands` [*number*] `... end`	Execute command list (group commands)
Control-C	Control-C	Interrupt the program
`#` [*text*]	`#` [*text*]	A comment
`am`	`am` (with `-xdb`), `set height` *num*	Activate more (turn on pagination)
`f` ["*printf-style-fmt*"]	No equivalent	Set address printing format
`h`	`h`	Help
`M[{t | c}` [*expr*[; *expr*...]]]	No equivalent	Print object or corefile map

`q`	`q`	Quit debugger
`sm`	`sm` (with `-xdb`), `set height 0`	Suspend more (turn off pagination)
`ss` *file*	No equivalent	Save (breakpoint, macro, assertion) state
`tc`	No equivalent	Toggle case sensitivity in searches

16.3 XDB Data Formats and HP WDB Equivalents

The format of the `print` command is different in XDB and GDB:

```
XDB:  p expr\fmt
GDB:  p/fmt expr
```

Use the GDB command `set print pretty` to obtain a structure display formatted similarly to the default XDB display.

XDB Command	HP WDB 2.0 Equivalent	Meaning
`b`	`d`	Byte in decimal
`B` (1)	`d`	Byte in decimal
`c`	`c`	Character
`C` (1)	`c`	Wide character
`d`	`d`	Decimal integer
`D` (1)	`d`	Long decimal integer
`e`	No equivalent	`e` floating-point notation as float
`E` (1)	No equivalent	`e` floating-point notation as double
`f`	No equivalent	`f` floating-point notation as float
`F` (1)	No equivalent	`f` floating-point notation as double
`g`	`f`	`g` floating-point notation as float
`G` (1)	`f`	`g` floating-point notation as double
`i`	Use `x/i` command	Machine instruction (disassembly)
`k`	No equivalent	Formatted structure display
`K` (1)	No equivalent	Formatted structure display with base classes

`n`	`print`	Normal (default) format, based on type
`o`	`o`	Expression in octal as integer
`O` (1)	`o`	Expression in octal as long integer
`p`	`a`	Print name of procedure containing address
`s`	No equivalent	String
`S`	No equivalent	Formatted structure display
`t`	`whatis`, `ptype`	Show type of the expression
`T` (1)	`ptype`	Show type of expression, including base class information
`u`	`u`	Expression in unsigned decimal format
`U` (1)	`u`	Expression in long unsigned decimal format
`w`	No equivalent	Wide character string
`W` (1)	No equivalent	Address of wide character string
`x`	`x`	Print in hexadecimal
`X` (1)	`x`	Print in long hexadecimal
`z`	`t`	Print in binary
`Z` (1)	`t`	Print in long binary

(1) HP WDB will display data in the size appropriate for the data. It will not extend the length displayed in response to one of the uppercase formchars (e.g. `O`, `D`, `F`).

16.4 XDB Location Syntax and HP WDB Equivalents

XDB Location Syntax	HP WDB 2.0 Equivalent	Meaning
line	*line*	Source line and code address
file[`:`*line*]	*file*[`:`*line*]	Source line and code address
proc	*proc*	Procedure name
[*file*`:`]*proc*[`:`*proc*[...]][`:`*line*]	No equivalent	Source line and code address
[*file*`:`]*proc*[`:`*proc*[...]][`:#`*label*]	No equivalent	Source line and code address
[*class*]`::`*proc*	[*class*]`::`*proc*	Source line and code address

[*class*]::*proc*[:*line*]	No equivalent	Source line and code address
[*class*]::*proc*[#*label*]	No equivalent	Source line and code address
proc#*line*	No equivalent	Code address
[*class*]::*proc*#*line*	No equivalent	Code address
#*label*	No equivalent	Source line and code address
name@*shared-library*	No equivalent	Address of *name* in shared library *shared-library*

16.5 XDB Special Language Operators and HP WDB Equivalents

XDB Language Operator	HP WDB 2.0 Equivalent	Meaning
`$addr`	Depends on language	Unary operator, address of object
`$in`	No equivalent	Unary Boolean operator, execution in procedure
`$sizeof`	`sizeof`	Unary operator, size of object

16.6 XDB Special Variables and HP WDB Equivalents

GDB does not provide special variables of the kind that XDB has, but you can use `show` and `set` to display and modify many debugger settings.

XDB Special Variable	HP WDB 2.0 Equivalent	Meaning
`$cplusplus`	No equivalent	C++ feature control flags
`$depth`	No equivalent	Default stack depth for local variables
`$fpa`	No equivalent	Treat FPA sequence as one instruction
`$fpa_reg`	No equivalent	Address register for FPA sequences
`$lang`	`show language`	Current language for expression evaluation
`$line`	No equivalent	Current source line number
`$malloc`	No equivalent	Debugger memory allocation (bytes)
`$print`	No equivalent	Display mode for character data
$*regname*	$*regname*	Hardware registers

`$result`	Use `$`*n* (value history number assigned to the desired result)	Return value of last command line procedure call
`$signal`	No equivalent	Current child procedure signal number
`$step`	No equivalent	Number of instructions debugger will step in non-debuggable procedures before free-running
`$`*var*	`$`*var*	Define or use special variable (convenience variable)

16.7 XDB Variable Identifiers and HP WDB Equivalents

XDB Variable Identifier	HP WDB 2.0 Equivalent	Meaning
var	*var*	Search for *var*
class`::`*var*	*class*`::`*var*	Search *class* for *var* (bug: not yet)
[[*class*]`::`]*proc*`:`[*class*`::`]*var*	*proc*`::`*var*	Search *proc* for *var* (static variables only)
[[*class*]`::`]*proc*`:`*depth*`:`[*class*`::`	No equivalent	Search *proc* for depth on stack
`. (dot)`	Empty string; for example, `p` is the equivalent of `p .`	Shorthand for last thing you looked at
`:`*var* or `::`*var*	`::`*var* to distinguish a global from a local variable with same name	Search for global variable only

16.8 Alphabetical Lists of XDB Commands and HP WDB Equivalents

16.8.1 A

XDB Command	Equivalent HP WDB 2.0 Command
`a` [*cmds*]	No equivalent
`aa` *number*	No equivalent
`aa *`	No equivalent
`ab` *number*	`enable` *number*
`ab *`	`enable`
`ab @`*shared-library*	No equivalent
`abc` *cmds*	No exact equivalent, but `display` *expr* is equivalent to `abc print` *expr*
`am`	`am` (with `-xdb`), `set height` *num*
`apm` *oldpath* [*newpath*]	No equivalent
`apm ""` [*newpath*]	No equivalent

16.8.2 B

XDB Command	Equivalent HP WDB 2.0 Command
`b` *loc*	`b` *loc*
`b`	`b`
`ba` *address*	`ba` *address* (with `-xdb`), `b *`*address*
`bb` [*depth*]	No equivalent (use `b` *proc*)
`bc` *number expr*	`bc` *number expr* (with `-xdb`), `ignore` *number expr* (within a commands list)
`bi` *expr*`.`*proc*	`b` *class*`::`*proc* `cond` *bnum* `(this ==` *expr*`)`
`bi -c` *expr*	No equivalent
`bi -C` *expr*	No equivalent
`bp`	No equivalent
`bp` *cmds*	No equivalent
`bpc -c` *class*	`rb ^`*class*`::*`
`bpc -C` *class*	Use `rb ^`*class*`::*` for base classes also
`bpo` *proc*	`rb` *proc*

`bpo` *class*`::`*proc*	`b` *class*`::`*proc*
`bpt`	No equivalent
`bpt` *cmds*	No equivalent
`bpx`	No equivalent
`bpx` *cmds*	No equivalent
`bt` [*depth*]	No equivalent

`bt` *proc*	`b` *proc* `commands` *bnum*
	`finish`
	`c`
	`end`

`bu` [*depth*]	`bu` [*depth*] (with `-xdb`). The `finish` command is equivalent to the sequence `bu`, `c`, `db` (to continue out of the current routine).
`bx` [*depth*]	`bx` [*depth*] (with `-xdb`)

16.8.3 C through D

XDB Command	Equivalent HP WDB 2.0 Command
`c`	`c`, `continue`
`c` *location*	`until` *location*
`C`	`c`, `continue`
`C` *location*	`until` *location*
`D "`*dir*`"`	`D "`*dir*`"` (with `-xdb`), `dir` *pathname*
`da` *number*	No equivalent
`da *`	No equivalent
`db`	`clear`
`db` *number*	`delete` *number*
`db *`	`delete`
`dbc`	`undisplay`
`def` *name replacement-text*	`def` *name* [GDB prompts for commands]
`down`	`up`
`down` *number*	`up` *number*
`dp`	No equivalent

`dpm` *index*	No equivalent
`dpm *`	No equivalent
`Dpt`	No equivalent
`Dpx`	No equivalent

16.8.4 F through K

<table>
<tr><th>XDB Command</th><th>Equivalent HP WDB 2.0 Command</th></tr>
<tr><td><code>f</code> ["printf-style-fmt"]</td><td>No equivalent</td></tr>
<tr><td><code>fr</code></td><td><code>fr</code> (with <code>-xdb -tui</code>), <code>display $fregs</code> (with <code>-tui</code>)</td></tr>
<tr><td><code>g</code> line</td><td><code>g</code> line (with <code>-xdb</code>), <code>go</code> line, <code>tb</code> line followed by <code>jump</code> line</td></tr>
<tr><td><code>g #</code>label</td><td>No equivalent</td></tr>
<tr><td><code>g {+ | -}</code>lines</td><td><code>g {+ | -}</code>lines (with <code>-xdb</code>), <code>go {+ | -}</code>lines <code>tb {+ | -}</code>lines followed by <code>jump {+ | -}</code>lines</td></tr>
<tr><td><code>g {+ | -}</code></td><td><code>g {+ | -}</code> (with <code>-xdb</code>), <code>go {+ | -}1</code>, <code>tb {+ | -}1</code> followed by <code>jump {+ | -}1</code></td></tr>
<tr><td><code>gr</code></td><td><code>gr</code> (with <code>-xdb -tui</code>), <code>display $regs</code> (with <code>-tui</code>)</td></tr>
<tr><td><code>h</code></td><td><code>h</code></td></tr>
<tr><td><code>if</code> expr {cmds} [{cmds}]</td><td><code>if</code> expr cmds [<code>else</code> cmds] <code>end</code></td></tr>
<tr><td><code>I</code></td><td><code>info</code> (many kinds), <code>show</code> (many kinds)</td></tr>
<tr><td><code>k</code></td><td><code>k</code></td></tr>
</table>

16.8.5 L

XDB Command	Equivalent HP WDB 2.0 Command
`l`	`l` (with `-xdb`), `info args` followed by `info locals`
`L`	`L` (with `-xdb`)
`la`	No equivalent
`lb`	`lb` (with `-xdb`), `i b`
`lc` [*string*]	`lc` [*string*] (with `-xdb`), `info common` *string*
`ld`	`ld` (with `-xdb`), `show directories`
`lf`	`lf` (with `-xdb`), `info sources`

`lf` [*string*]	No equivalent
`lg` [*string*]	`lg` [*string*] (with `-xdb`), `info variables` [*string*]
`ll` [*string*]	`info functions` [*string*], `info variables` [*string*], `maint print msymbols` *file*
`lm` [*string*]	`show user` [*string*]
`lo` [[*class*]`::`][*string*]	`info func` [[*class*]`::`][*string*]
`lp`	`info functions`
`lp` [[*class*]`::`]*string*	`info func` [[*class*]`::`]*string* `info addr` [[*class*]`::`]*string*
`lpm`	No equivalent
`lr`	`lr` (with `-xdb`), `info all-reg`
`lr` *string*	`lr` *string* (with `-xdb`), `info reg` *string*
`ls` [*string*]	No equivalent
`lz`	`lz` (with `-xdb`), `info signals`

16.8.6 M through P

XDB Command	Equivalent HP WDB 2.0 Command
`M`[`{t | c}` [*expr*[`;` *expr*...]]]	No equivalent
`mm`	`info sharedlibrary`
`mm` *string*	No equivalent
`N`	`fo` or `rev`
`n`	`fo` or `rev`
`p` *expr*[*format*]	`p`[`/`*format*] *expr* [Note: The *count* and *size* portions of formats are not allowed in the `p` (`print`) command. They are allowed in the `x` command (examine memory).]
`p` *expr*`?`*format*	`p/`*format* `&`*expr*
`p` *class*`::`	No equivalent
`p $lang`	`show language`
`p {+ | -}`[*format*	Use `x/`*format* command to obtain initial value, then use `x` with no argument to obtain value of next memory location. To obtain value of previous memory location, use `"x $_ - 1"`.
`pq` *expr*	`set` *expr*, `set var` *expr*

`pq` *expr?format*	No equivalent
`pq` *class*`::`	No equivalent
`pq` [`+	-`][`\`*format*	No equivalent

16.8.7 Q through S

XDB Command	Equivalent HP WDB 2.0 Command
`q`	`q`
`Q`	`Q` (with `-xdb`), `silent` (must be first command in a commands list)
`r` [*arguments*]	`r` [*arguments*]
`R`	`R` (with `-xdb`), `r`
`s`	`s`, `si`
`s` *number*	`s` *number*, `si` *number*
`S`	`S` (with `-xdb`), `n`, `ni`
`S` *number*	`S` *number* (with `-xdb`), `n` *number*, `ni`*number*
`sa` *number*	No equivalent
`sa *`	No equivalent
`sb` *number*	`disable` *number*
`sb *`	`disable`
`sb @`*shared-library*	No equivalent
`sm`	`sm` (with `-xdb`), `set height 0`
`sr`	`sr` (with `-xdb -tui`), `display $sregs` (with `-tui`)
`ss` *file*	No equivalent

16.8.8 T

XDB Command	Equivalent HP WDB 2.0 Command
`t` [*depth*]	`t` [*depth*] (with `-xdb`), `bt` [*depth*]
`T` [*depth*]	`T` [*depth*] (with `-xdb`), `bt full` [*depth*]
`ta`	No equivalent
`tb`	No equivalent

`tc`	No equivalent
`td`	`td` (with `-xdb -tui`)
`tf`	`tf` (with `-xdb -tui`), `toggle $fregs` (with `-tui`)
`tm`	No equivalent
`top`	`frame 0`
`tr` [`@`]	No equivalent
`ts`	`ts` (with `-xdb -tui`)

16.8.9 U through Z

XDB Command	Equivalent HP WDB 2.0 Command
`u`	`u` (with `-xdb -tui`), `update` (with `-tui`)
`U`	`U` (with `-xdb -tui`), `refresh` (with `-tui`)
`undef` *name*	`def` *name* [follow with empty command list]
`undef *`	No equivalent
`up`	`down`
`up` *number*	`down` *number*
`v`	`v` (with `-xdb`), `list`
`v` *location*	`v` *location* (with `-xdb`), `list` *location*
`V` [*depth*]	`V` [*depth*] (with `-xdb`), `frame` [*depth*]
`va` *address*	`va` *address* (with `-xdb`), `disas` *address*
`va` *label*	`va` *label* (with `-xdb`), `disas` *label*
`va` *label* `+` *offset*	`va` *label* `+` *offset* (with `-xdb`), `disas` *label* `+` *offset*
`w` *number*	`w` *number* (with `-xdb -tui`), `winheight src` *number* (with `-tui`)
`x` [*expr*]	No equivalent
`xdb` *program*	`gdb -xdb` *program*, `gdb -xdb -tui` *program*
`xdb` *program corefile*	`gdb -xdb` *program* `-c` *corefile*
`xdb -d` *dir*	`gdb -xdb -d` *dir*
`xdb -P` *pid program*	`gdb -xdb` *program pid*
`xdb -i`	(after starting) `run <` *file*

`xdb -o`	(after starting) `run >` *file*
`z` *number* `s`	`z` *number* `s` (with `-xdb`), `handle` *number* `stop`, `handle` *number* `nostop`
`z` *number* `i`	`z` *number* `i` (with `-xdb`), `handle` *number* `nopass`, `handle` *number* `pass`
`z` *number* `r`	`z` *number* `r` (with `-xdb`), `handle` *number* `print`, `handle` *number* `noprint`
`z` *number* `Q`	`z` *number* `Q` (with `-xdb`), `handle` *number* `noprint`

16.8.10 Symbols

XDB Symbol	Equivalent HP WDB Symbol
line	*line*
file[`:`*line*]	*file*[`:`*line*]
proc	*proc*
[*file*`:`]*proc*[`:`*proc*[...]][`:`*line*]	No equivalent
[*file*`:`]*proc*[`:`*proc*[...]][`:#`*label*]	No equivalent
[*class*]`::`*proc*	[*class*]`::`*proc*
[*class*]`::`*proc*[`:`*line*]	No equivalent
[*class*]`::`*proc*[`#`*label*]	No equivalent
proc`#`*line*	No equivalent
[*class*]`::`*proc*`#`*line*	No equivalent
name`@`*shared-library*	No equivalent
var	*var*
class`::`*var*	*class*`::`*var* (bug: not yet)
[[*class*]`::`]*proc*`:`[*class*`::`]*var*	*proc*`::`*var* (static variables only)
[[*class*]`::`]*proc*`:`*depth*`:`[*class*`::`]*var*	No equivalent
Return	*Return*
`"`*any_string*`"`	p `"`*any_string*`"`
`.` (dot)	Empty string; for example, `p` is the equivalent of `p .`
`~`	*Return*

{+ | -}r
: {+ | -}r (with -xdb -tui), {+ | -} data (with -tui)

{+ | -}[*number*]
: {+ | -}[*number*] (with -tui; note that a space is required between + or - and the *number*)

/[*string*]
: /[*string*] (with -xdb), search *regexp*, forw *regexp*

?[*string*]
: ?[*string*] (with -xdb), rev *regexp*

;
: No equivalent (one command per line in command list)

:*var* or ::*var*
: ::*var*

! *cmd_line*
: ! *cmd_line* (with -xdb), she *cmd_line*

{*cmd_list*}
: commands [*number*] ... end

<*file*
: source *file*

<<*file*
: No equivalent

>
: No equivalent

>*file*
: No equivalent

>c
: No equivalent

>f
: No equivalent

>t
: No equivalent

>@[c | f | t]
: No equivalent

>@*file*
: No equivalent

>>
: No equivalent

>>*file*
: No equivalent

>>@
: No equivalent

>>@*file*
: No equivalent

Control-C
: Control-C

[*text*]
: # [*text*]

#*label*
: No equivalent

17 Controlling GDB

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, see Section 8.7 [Print settings], page 70. Other settings are described here.

17.1 Prompt

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

Note: `set prompt` does not add a space for you after the prompt you set. This allows you to set a prompt which ends in a space or a prompt that does not.

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

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

17.2 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, `csh`-like history substitution, and a storage and recall of command history across debugging sessions.

You may control the behavior of command line editing in GDB with the command `set`.

`set editing`
`set editing on`
: Enable command line editing (enabled by default).

`set editing off`
: Disable command line editing.

`show editing`
: Show whether command line editing is enabled.

17.3 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 these commands to manage the GDB command history facility.

To make command history understand your vi key bindings you need to create a '`~/.inputrc`' file with the following contents:

```
set editing-mode vi
```

The *readline* interface uses the '`.inputrc`' file to control the settings.

`set history filename` *fname*

Set the name of the GDB command history file to *fname*. This is the file where GDB reads an initial command history list, and where it writes the command history from this session when it exits. You can access this list through history expansion or through the history command editing characters listed below. This file defaults to the value of the environment variable `GDBHISTFILE`, or to '`./.gdb_history`' ('`./_gdb_history`' on MS-DOS) if this variable is not set.

`set history save`
`set history save on`

Record command history in a file, whose name may be specified with the `set history filename` command. By default, this option is disabled.

`set history save off`

Stop recording command history in a file.

`set history size` *size*

Set the number of commands which GDB keeps in its history list. This defaults to the value of the environment variable `HISTSIZE`, or to 256 if this variable is not set.

History expansion assigns special meaning to the character `!`.

Since `!` is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the `set history expansion on` command, you may sometimes need to follow `!` (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities do not attempt substitution on the strings `!=` and `!(`, even when history expansion is enabled.

The commands to control history expansion are:

`set history expansion on`
`set history expansion`

Enable history expansion. History expansion is off by default.

`set history expansion off`

Disable history expansion.

The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with GNU Emacs or `vi` may wish to read it.

`show history`
`show history filename`
`show history save`
`show history size`
`show history expansion`

These commands display the state of the GDB history parameters. `show history` by itself displays all four states.

`show commands`

Display the last ten commands in the command history.

`show commands` *n*
: Print ten commands centered on command number *n*.

`show commands +`
: Print ten commands just after the commands last printed.

17.4 Screen size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type ⟨RET⟩ when you want to continue the output, or `q` to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally GDB knows the size of the screen from the terminal driver software. For example, on Unix GDB uses the termcap data base together with the value of the `TERM` environment variable and the `stty rows` and `stty cols` settings. If this is not correct, you can override it with the `set height` and `set width` commands:

`set height` *lpp*
`show height`
`set width` *cpl*
`show width`
: These `set` commands specify a screen height of *lpp* lines and a screen width of *cpl* characters. The associated `show` commands display the current settings.

  If you specify a height of zero lines, GDB does not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer.

  Likewise, you can specify '`set width 0`' to prevent GDB from wrapping its output.

17.5 Numbers

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with '`0`', decimal numbers end with '`.`', and hexadecimal numbers begin with '`0x`'. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers—when no particular format is specified—is base 10. You can change the default base for both input and output with the `set radix` command.

`set input-radix` *base*
: Set the default base for numeric input. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix; for example, any of

```
set radix 012
set radix 10.
set radix 0xa
```

sets the base to decimal. On the other hand, '`set radix 10`' leaves the radix unchanged no matter what it was.

`set output-radix` *base*
: Set the default base for numeric display. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix.

`show input-radix`
: Display the current default base for numeric input.

`show output-radix`
: Display the current default base for numeric display.

17.6 Optional warnings and messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the `set verbose` command. This makes GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by `set verbose` are those which announce that the symbol table for a source file is being read; see `symbol-file` in Section 12.1 [Commands to specify files], page 109.

`set verbose on`
: Enables GDB output of certain informational messages.

`set verbose off`
: Disables GDB output of certain informational messages.

`show verbose`
: Displays whether `set verbose` is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful (see Section 12.2 [Errors reading symbol files], page 113).

`set complaints` *limit*
: Permits GDB to output *limit* complaints about each type of unusual symbols before becoming silent about the problem. Set *limit* to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

`show complaints`
: Displays how many symbol complaints GDB is permitted to produce.

By default, GDB is cautious, and asks what sometimes seems to be a lot of stupid questions to confirm certain commands. For example, if you try to run a program which is already running:

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

If you are willing to unflinchingly face the consequences of your own commands, you can disable this "feature":

`set confirm off`
: Disables confirmation requests.

`set confirm on`
: Enables confirmation requests (the default).

`show confirm`
: Displays state of confirmation requests.

18 Canned Sequences of Commands

Aside from breakpoint commands (see Section 5.1.7 [Breakpoint command lists], page 42), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

18.1 User-defined commands

A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command. This is done with the `define` command. User commands may accept up to 10 arguments separated by white space. Arguments are accessed within the user command via *$arg0...$arg9*. A trivial example:

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

To execute the command use:

```
adder 1 2 3
```

This defines the command `adder`, which prints the sum of its three arguments. Note the arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior functions calls.

`define` *commandname*
: Define a command named *commandname*. If there is already a command by that name, you are asked to confirm that you want to redefine it.

  The definition of the command is made up of other GDB command lines, which are given following the `define` command. The end of these commands is marked by a line containing `end`.

`if`
: Takes a single argument, which is an expression to evaluate. It is followed by a series of commands that are executed only if the expression is true (nonzero). There can then optionally be a line `else`, followed by a series of commands that are only executed if the expression was false. The end of the list is marked by a line containing `end`.

`while`
: The syntax is similar to `if`: the command takes a single argument, which is an expression to evaluate, and must be followed by the commands to execute, one per line, terminated by an `end`. The commands are executed repeatedly as long as the expression evaluates to true.

`document` *commandname*
: Document the user-defined command *commandname*, so that it can be accessed by `help`. The command *commandname* must already be defined. This command reads lines of documentation just as `define` reads the lines of the command definition, ending with `end`. After the `document` command is finished, `help` on command *commandname* displays the documentation you have written.

  You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

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

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

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

18.2 User-defined command hooks

You may define *hooks*, which are a special kind of user-defined command. Whenever you run the command '`foo`', if the user-defined command '`hook-foo`' exists, it is executed (with no arguments) before that command.

In addition, a pseudo-command, '`stop`' exists. Defining ('`hook-stop`') makes the associated commands execute every time execution stops in your program: before breakpoint commands are run, displays are printed, or the stack frame is printed.

For example, to ignore `SIGALRM` signals while single-stepping, but treat them normally during normal execution, you could define:

```
define hook-stop
handle SIGALRM nopass
end

define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

You can define a hook for any single-word command in GDB, 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 typed 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.

18.3 Command files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting with `#`) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When you start GDB, it automatically executes commands from its *init files*. These are files named '`.gdbinit`' on Unix, or '`gdb.ini`' on DOS/Windows. GDB reads the init file (if any) in your home directory[1], 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 Section 2.1.2 [Choosing modes], page 11.

It can be useful to create a '`.gdbinit`' file in the directory where you are debugging an application. This file will set the set of actions that apply for this application.

For example, one might add lines like:

```
dir /usr/src/path/to/source/files
```

to add source directories or:

```
break fatal
```

to set breakpoints on fatal error routines or diagnostic routines.

On some configurations of GDB, the init file is known by a different name (these are typically environments where a specialized form of GDB may need to coexist with other forms, hence a different name for the specialized version's init file). These are the environments with special init file names:

- VxWorks (Wind River Systems real-time OS): '`.vxgdbinit`'
- OS68K (Enea Data Systems real-time OS): '`.os68gdbinit`'
- ES-1800 (Ericsson Telecom AB M68000 emulator): '`.esgdbinit`'

You can also request the execution of a command file with the `source` command:

`source` *filename*
: Execute the command file *filename*.

The lines in a command file are executed sequentially. They are not printed as they are executed. An error in any command terminates execution of the command file.

Commands that would ask for confirmation if used interactively proceed without asking when used in a command file. Many GDB commands that normally print messages to say what they are doing omit the messages when called from command files.

---

[1] On DOS/Windows systems, the home directory is the one pointed to by the `HOME` environment variable.

18.4 Commands for controlled output

During the execution of a command file or a user-defined command, normal GDB output is suppressed; the only output that appears is what is explicitly printed by the commands in the definition. This section describes three commands useful for generating exactly the output you want.

`echo` *text*
: Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as '`\n`' to print a newline. **No newline is printed unless you specify one.** In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments. To print '` and foo = `', use the command '`echo \ and foo = \ `'.

A backslash at the end of *text* can be used, as in C, to continue the command onto subsequent lines. For example,

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

produces the same output as

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

`output` *expression*
: Print the value of *expression* and nothing but that value: no newlines, no '`$`*nn* `= `'. The value is not entered in the value history either. See Section 8.1 [Expressions], page 63, for more information on expressions.

`output/`*fmt expression*
: Print the value of *expression* in format *fmt*. You can use the same formats as for `print`. See Section 8.4 [Output formats], page 66, for more information.

`printf` *string*`,` *expressions*`...`
: Print the values of the *expressions* under the control of *string*. The *expressions* are separated by commas and may be either numbers or pointers. Their values are printed as specified by *string*, exactly as if your program were to execute the C subroutine

```
printf (string, expressions...);
```

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences that you can use in the format string are the simple ones that consist of backslash followed by a letter.

19 Using GDB under GNU Emacs

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

To use this interface, use the command *M-x gdb* in Emacs. Give the executable file you want to debug as an argument. This command starts GDB as a subprocess of Emacs, with input and output through a newly created Emacs buffer. (Do not use the `-tui` option to run GDB from Emacs.)

Using GDB under Emacs is just like using GDB normally except for two things:

- All "terminal" input and output goes through the Emacs buffer.

This applies both to GDB commands and their output, and to the input and output done by the program you are debugging.

This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

All the facilities of Emacs' Shell mode are available for interacting with your program. In particular, you can send signals the usual way—for example, *C-c C-c* for an interrupt, *C-c C-z* for a stop.

- GDB displays source code through Emacs.

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

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

> *Warning:* If the directory where your program resides is not your current directory, it can be easy to confuse Emacs about the location of the source files, in which case the auxiliary display buffer does not appear to show your source. GDB can find programs by searching your environment's `PATH` variable, so the GDB input and output session proceeds normally; but Emacs does not get enough information back from GDB to locate the source files in this situation. To avoid this problem, either start GDB mode from the directory where your program resides, or specify an absolute file name when prompted for the *M-x gdb* argument.
>
> A similar confusion can result if you use the GDB `file` command to switch to debugging a program in some other location, from an existing GDB buffer in Emacs.

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

```
(setq gdb-command-name "mygdb")
```

(preceded by *M-:* or *ESC :*, or typed in the `*scratch*` buffer, or in your '`.emacs`' file) makes Emacs call the program named "`mygdb`" instead.

In the GDB I/O buffer, you can use these special Emacs commands in addition to the standard Shell mode commands:

C-h m Describe the features of Emacs' GDB Mode.

M-s Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

M-n Execute to next source line in this function, skipping all function calls, like the GDB `next` command. Then update the display window to show the current file and location.

M-i Execute one instruction, like the GDB `stepi` command; update display window accordingly.

M-x gdb-nexti
Execute to next instruction, using the GDB `nexti` command; update display window accordingly.

C-c C-f Execute until exit from the selected stack frame, like the GDB `finish` command.

M-c Continue execution of your program, like the GDB `continue` command.
Warning: In Emacs v19, this command is *C-c C-p*.

M-u Go up the number of frames indicated by the numeric argument (see section "Numeric Arguments" in *The* GNU *Emacs Manual*), like the GDB `up` command.
Warning: In Emacs v19, this command is *C-c C-u*.

M-d Go down the number of frames indicated by the numeric argument, like the GDB `down` command.
Warning: In Emacs v19, this command is *C-c C-d*.

C-x & Read the number where the cursor is positioned, and insert it at the end of the GDB I/O buffer. For example, if you wish to disassemble code around an address that was displayed earlier, type *disassemble*; then move the cursor to the address display, and pick up the argument for `disassemble` by typing *C-x &*.
You can customize this further by defining elements of the list `gdb-print-command`; once it is defined, you can format or otherwise process numbers picked up by *C-x &* before they are inserted. A numeric argument to *C-x &* indicates that you wish special formatting, and also acts as an index to pick an element of the list. If the list element is a string, the number to be inserted is formatted using the Emacs function `format`; otherwise the number is passed as an argument to the corresponding list element.

In any source file, the Emacs command *C-x SPC* (`gdb-break`) tells GDB to set a breakpoint on the source line point is on.

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command `f` in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers. If you add or delete lines from the text, the line numbers that GDB knows cease to correspond properly with the code.

20 Reporting Bugs in GDB

Your bug reports play an essential role in making GDB reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of GDB work better. Bug reports are your contribution to the maintenance of GDB.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

20.1 Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the debugger gets a fatal signal, for any input whatever, that is a GDB bug. Reliable debuggers never crash.
- If GDB produces an error message for valid input, that is a bug. (Note that if you're cross debugging, the problem may also be somewhere in the connection to the target.)
- If GDB does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of debugging tools, your suggestions for improvement of GDB are welcome in any case.

20.2 How to report bugs

If you obtained GDB (Hewlett-Packard Wildebeest (based on GDB 4.17-hpwdb-980821)) as part of your HP ANSI C, HP ANSI C++, or HP Fortran compiler kit, report problems to your HP Support Representative.

If you obtained GDB (Hewlett-Packard Wildebeest (based on GDB 4.17-hpwdb-980821)) from the Hewlett-Packard Web site, report problems to your HP Support Representative. Support is covered under the support contract for your HP compiler.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the debugger into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug. It may be that the bug has been reported previously, but neither you nor we can know that unless your bug report is complete and self-contained.

Sometimes people give a few sketchy facts and ask, “Does this ring a bell?” Those bug reports are useless, and we urge everyone to *refuse to respond to them* except to chide the sender to report bugs properly.

To enable us to fix the bug, you should include all these things:

- The version of GDB. GDB announces it if you start with no arguments; you can also print it at any time using `show version`.

  Without this, we will not know whether there is any point in looking for the bug in the current version of GDB.

- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile the program you are debugging— e.g. “HP92453-01 A.10.32.03 HP C Compiler”. Use the `what` command with the pathname of the compile command (‘`what /opt/ansic/bin/cc`’, for example) to obtain this information.
- The command arguments you gave the compiler to compile your example and observe the bug. For example, did you use ‘`-O`’? To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

  If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.

- A complete input script, and all necessary source files, that will reproduce the bug.
- A description of what behavior you observe that you believe is incorrect. For example, “It gets a fatal signal.”

  Of course, if the bug is that GDB gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of GDB is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

  However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

  A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GDB it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

  And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

21 Command Line Editing

This chapter describes the basic features of the GNU command line editing interface.

21.1 Introduction to Line Editing

The following paragraphs describe the notation used to represent keystrokes.

The text ⟨C-k⟩ is read as 'Control-K' and describes the character produced when the ⟨k⟩ key is pressed while the Control key is depressed.

The text ⟨M-k⟩ is read as 'Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the ⟨k⟩ key is pressed. If you do not have a meta key, the identical keystroke can be generated by typing ⟨ESC⟩ *first*, and then typing ⟨k⟩. Either process is known as *metafying* the ⟨k⟩ key.

The text ⟨M-C-k⟩ is read as 'Meta-Control-k' and describes the character produced by *metafying* ⟨C-k⟩.

In addition, several keys have their own names. Specifically, ⟨DEL⟩, ⟨ESC⟩, ⟨LFD⟩, ⟨SPC⟩, ⟨RET⟩, and ⟨TAB⟩ all stand for themselves when seen in this text, or in an init file (see Section 21.3 [Readline Init File], page 193).

21.2 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 press ⟨RETURN⟩. You do not have to be at the end of the line to press ⟨RETURN⟩; the entire line is accepted regardless of the location of the cursor within the line.

21.2.1 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 your erase character 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 type ⟨C-b⟩ to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with ⟨C-f⟩.

When you add text in the middle of a line, you will notice that characters to the right of the cursor are '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 are 'pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

⟨C-b⟩ Move back one character.

⟨C-f⟩ Move forward one character.

⟨DEL⟩ Delete the character to the left of the cursor.

⟨C-d⟩ Delete the character underneath the cursor.

Printing characters
: Insert the character into the line at the cursor.

⟨C-_⟩ Undo the last editing command. You can undo all the way back to an empty line.

21.2.2 Readline Movement Commands

The above table describes 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 ⟨C-b⟩, ⟨C-f⟩, ⟨C-d⟩, and ⟨DEL⟩. Here are some commands for moving more rapidly about the line.

⟨C-a⟩ Move to the start of the line.

⟨C-e⟩ Move to the end of the line.

⟨M-f⟩ Move forward a word, where a word is composed of letters and digits.

⟨M-b⟩ Move backward a word.

⟨C-l⟩ Clear the screen, reprinting the current line at the top.

Notice how ⟨C-f⟩ moves forward a character, while ⟨M-f⟩ moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

21.2.3 Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* (re-inserting) 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.

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 all. 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.

Here is the list of commands for killing text.

⟨C-k⟩ Kill the text from the current cursor position to the end of the line.

⟨M-d⟩ Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

⟨M-DEL⟩ Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.

⟨C-w⟩ Kill from the cursor to the previous whitespace. This is different than ⟨M-DEL⟩ because the word boundaries differ.

Here is how to *yank* the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

⟨C-y⟩ Yank the most recently killed text back into the buffer at the cursor.

⟨M-y⟩ Rotate the kill-ring, and yank the new top. You can only do this if the prior command is ⟨C-y⟩ or ⟨M-y⟩.

21.2.4 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type '`M-- C-k`'.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' typed is a minus sign (⟨-⟩), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the ⟨C-d⟩ command an argument of 10, you could type '`M-1 0 C-d`'.

21.2.5 Searching for Commands in the History

Readline provides commands for searching through the command history for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, Readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. The characters present in the value of the *isearch-terminators* variable are used to terminate an incremental search. If that variable has not been assigned a value, the ⟨ESC⟩ and ⟨C-J⟩ characters will terminate an incremental search. ⟨C-g⟩ will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line. To find other matching entries in the history list, type ⟨C-s⟩ or ⟨C-r⟩ as appropriate. This will search backward or forward in the history for the next entry matching the search string typed so far. Any other key sequence bound to a Readline command will terminate the search and execute that command. For instance, a ⟨RET⟩ will terminate the search and accept the line, thereby executing the command from the history list.

Non-incremental searches read the entire search string before starting to search for matching history lines. The search string may be typed by the user or be part of the contents of the current line.

21.3 Readline Init File

Although the Readline library comes with a set of `emacs`-like keybindings installed by default, it is possible to use a different set of keybindings. Any user can customize programs that use Readline by putting commands in an *inputrc* file in his home directory. The name of this file is taken from the value of the environment variable `INPUTRC`. If that variable is unset, the default is '`~/.inputrc`'.

When a program which uses the Readline library starts up, the init file is read, and the key bindings are set.

In addition, the `C-x C-r` command re-reads this init file, thus incorporating any changes that you might have made to it.

21.3.1 Readline Init File Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a '`#`' are comments. Lines beginning with a '`$`' indicate conditional constructs (see Section 21.3.2 [Conditional Init Constructs], page 198). Other lines denote variable settings and key bindings.

Variable Settings

You can modify the run-time behavior of Readline by altering the values of variables in Readline using the `set` command within the init file. Here is how to change from the default Emacs-like key binding to use `vi` line editing commands:

```
set editing-mode vi
```

A great deal of run-time behavior is changeable with the following variables.

`bell-style`

Controls what happens when Readline wants to ring the terminal bell. If set to '`none`', Readline never rings the bell. If set to '`visible`', Readline uses a visible bell if one is available. If set to '`audible`' (the default), Readline attempts to ring the terminal's bell.

`comment-begin`

The string to insert at the beginning of the line when the `insert-comment` command is executed. The default value is `"#"`.

`completion-ignore-case`

If set to '`on`', Readline performs filename matching and completion in a case-insensitive fashion. The default value is '`off`'.

`completion-query-items`

The number of possible completions that determines when the user is asked whether he wants to see the list of possibilities. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. The default limit is `100`.

`convert-meta`

If set to '`on`', Readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prepending an ⟨ESC⟩ character, converting them to a meta-prefixed key sequence. The default value is '`on`'.

`disable-completion`
: If set to '`On`', Readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to `self-insert`. The default is '`off`'.

`editing-mode`
: The `editing-mode` variable controls which default set of key bindings is used. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can be set to either '`emacs`' or '`vi`'.

`enable-keypad`
: When set to '`on`', Readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys. The default is '`off`'.

`expand-tilde`
: If set to '`on`', tilde expansion is performed when Readline attempts word completion. The default is '`off`'.

`horizontal-scroll-mode`
: This variable can be set to either '`on`' or '`off`'. Setting it to '`on`' means that the text of the lines being edited will scroll horizontally on a single screen line when they are longer than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to '`off`'.

`input-meta`
: If set to '`on`', Readline will enable eight-bit input (it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The default value is '`off`'. The name `meta-flag` is a synonym for this variable.

`isearch-terminators`
: The string of characters that should terminate an incremental search without subsequently executing the character as a command (see Section 21.2.5 [Searching], page 193). If this variable has not been given a value, the characters ⟨ESC⟩ and ⟨C-J⟩ will terminate an incremental search.

`keymap`
: Sets Readline's idea of the current keymap for key binding commands. Acceptable `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`mark-directories`
: If set to '`on`', completed directory names have a slash appended. The default is '`on`'.

`mark-modified-lines`
: This variable, when set to '`on`', causes Readline to display an asterisk ('`*`') at the start of history lines which have been modified. This variable is '`off`' by default.

`output-meta`
: If set to '`on`', Readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is '`off`'.

`print-completions-horizontally`
: If set to '`on`', Readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen. The default is '`off`'.

`show-all-if-ambiguous`
: This alters the default behavior of the completion functions. If set to '`on`', words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell. The default value is '`off`'.

`visible-stats`
: If set to '`on`', a character denoting a file's type is appended to the filename when listing possible completions. The default is '`off`'.

Key Bindings

The syntax for controlling key bindings in the init file is simple. First you have to know the name of the command that you want to change. The following sections contain tables of the command name, the default keybinding, if any, 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 init 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 above example, ⟨C-u⟩ is bound to the function `universal-argument`, and ⟨C-o⟩ is bound to run the macro expressed on the right hand side (that is, to insert the text '`> output`' into the line).

"*keyseq*": *function-name* or *macro*
: *keyseq* differs from *keyname* above in that strings denoting an entire key sequence can be specified, by placing the key sequence in double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the special character names are not recognized.

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

In the above example, C-u is bound to the function `universal-argument` (just as it was in the first example), ‘C-x C-r’ is bound to the function `re-read-init-file`, and ‘ESC [1 1 ~’ is bound to insert the text ‘`Function Key 1`’.

The following GNU Emacs style escape sequences are available when specifying key sequences:

`\C-` control prefix

`\M-` meta prefix

`\e` an escape character

`\\` backslash

`\"` "

`\'` '

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

`\a` alert (bell)

`\b` backspace

`\d` delete

`\f` form feed

`\n` newline

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\`*nnn* the character whose ASCII code is the octal value *nnn* (one to three digits)

`\x`*nnn* the character whose ASCII code is the hexadecimal value *nnn* (one to three digits)

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes described above are expanded. Backslash will quote any other character in the macro text, including ‘"’ and ‘'’. For example, the following binding will make ‘`C-x \`’ insert a single ‘`\`’ into the line:

```
"\C-x\\": "\\"
```

21.3.2 Conditional Init Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are four parser directives used.

`$if` The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test extends to the end of the line; no characters are required to isolate it.

- `mode` The `mode=` form of the `$if` directive is used to test whether Readline is in `emacs` or `vi` mode. This may be used in conjunction with the '`set keymap`' command, for instance, to set bindings in the `emacs-standard` and `emacs-ctlx` keymaps only if Readline is starting out in `emacs` mode.

- `term` The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the '`=`' is tested against both the full name of the terminal and the portion of the terminal name before the first '`-`'. This allows `sun` to match both `sun` and `sun-cmd`, for instance.

- `application`
  The *application* construct is used to include application-specific settings. Each program using the Readline library sets the *application name*, and you can test for it. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

`$endif` This command, as seen in the previous example, terminates an `$if` command.

`$else` Commands in this branch of the `$if` directive are executed if the test fails.

`$include` This directive takes a single filename as an argument and reads commands and bindings from that file.

```
$include /etc/inputrc
```

21.3.3 Sample Init File

Here is an example of an inputrc file. This illustrates key binding, variable assignment, and conditional syntax.

```
# This file controls the behaviour of line input editing for
# programs that use the Gnu Readline library.  Existing programs
# include FTP, Bash, and Gdb.
#
# You can re-read the inputrc file with C-x C-r.
# Lines beginning with '#' are comments.
#
# First, include any systemwide bindings and variable assignments from
# /etc/Inputrc
$include /etc/Inputrc

#
# Set various bindings for emacs mode.

set editing-mode emacs

$if mode=emacs

Meta-Control-h:  backward-kill-word Text after the function name is ignored

#
# Arrow keys in keypad mode
#
#"\M-OD":        backward-char
#"\M-OC":        forward-char
#"\M-OA":        previous-history
#"\M-OB":        next-history
#
# Arrow keys in ANSI mode
#
"\M-[D":        backward-char
"\M-[C":        forward-char
"\M-[A":        previous-history
"\M-[B":        next-history
#
# Arrow keys in 8 bit keypad mode
#
#"\M-\C-OD":       backward-char
#"\M-\C-OC":       forward-char
#"\M-\C-OA":       previous-history
#"\M-\C-OB":       next-history
#
# Arrow keys in 8 bit ANSI mode
#
#"\M-\C-[D":       backward-char
#"\M-\C-[C":       forward-char
#"\M-\C-[A":       previous-history
#"\M-\C-[B":       next-history
```

```
C-q: quoted-insert

$endif

# An old-style binding.  This happens to be the default.
TAB: complete

# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word -- insert open and close double quotes
# and move to just after the open quote
"\C-x\"": "\"\"\C-b"
# insert a backslash (testing backslash escapes in sequences and macros)
"\C-x\\": "\\"
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
# Add a binding to refresh the line, which is unbound
"\C-xr": redraw-current-line
# Edit variable on current line.
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# use a visible bell if one is available
set bell-style visible

# don't strip characters to 7 bits when reading
set input-meta on

# allow iso-latin1 characters to be inserted rather than converted to
# prefix-meta sequences
set convert-meta off

# display characters with the eighth bit set directly rather than
# as meta-prefixed characters
set output-meta on

# if there are more than 150 possible completions for a word, ask the
# user if he wants to see all of them
set completion-query-items 150

# For FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
$endif
```

21.4 Bindable Readline Commands

This section describes Readline commands that may be bound to key sequences.

21.4.1 Commands For Moving

`beginning-of-line (C-a)`
: Move to the start of the current line.

`end-of-line (C-e)`
: Move to the end of the line.

`forward-char (C-f)`
: Move forward a character.

`backward-char (C-b)`
: Move back a character.

`forward-word (M-f)`
: Move forward to the end of the next word. Words are composed of letters and digits.

`backward-word (M-b)`
: Move back to the start of this, or the previous, word. Words are composed of letters and digits.

`clear-screen (C-l)`
: Clear the screen and redraw the current line, leaving the current line at the top of the screen.

`redraw-current-line ()`
: Refresh the current line. By default, this is unbound.

21.4.2 Commands For Manipulating The History

`accept-line (Newline, Return)`
: Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)`
: Move 'up' through the history list.

`next-history (C-n)`
: Move 'down' through the history list.

`beginning-of-history (M-<)`
: Move to the first line in the history.

`end-of-history (M->)`
: Move to the end of the input history, i.e., the line currently being entered.

`reverse-search-history (C-r)`
: Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)`
: Search forward starting at the current line and moving 'down' through the the history as necessary. This is an incremental search.

`non-incremental-reverse-search-history (M-p)`
: Search backward starting at the current line and moving 'up' through the history as necessary using a non-incremental search for a string supplied by the user.

`non-incremental-forward-search-history (M-n)`
: Search forward starting at the current line and moving 'down' through the the history as necessary using a non-incremental search for a string supplied by the user.

`history-search-forward ()`
: Search forward through the history for the string of characters between the start of the current line and the current cursor position (the *point*). This is a non-incremental search. By default, this command is unbound.

`history-search-backward ()`
: Search backward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search. By default, this command is unbound.

`yank-nth-arg (M-C-y)`
: Insert the first argument to the previous command (usually the second word on the previous line). With an argument *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command.

`yank-last-arg (M-., M-_)`
: Insert last argument to the previous command (the last word of the previous history entry). With an argument, behave exactly like `yank-nth-arg`. Successive calls to `yank-last-arg` move back through the history list, inserting the last argument of each line in turn.

21.4.3 Commands For Changing Text

`delete-char (C-d)`
: Delete the character under the cursor. If the cursor is at the beginning of the line, there are no characters in the line, and the last character typed was not bound to `delete-char`, then return `EOF`.

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric argument means to kill the characters instead of deleting them.

`forward-backward-delete-char ()`
: Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cursor is deleted. By default, this is not bound to a key.

`quoted-insert (C-q, C-v)`
: Add the next character typed to the line verbatim. This is how to insert key sequences like C-q, for example.

`tab-insert (M-TAB)`
: Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`
: Insert yourself.

`transpose-chars (C-t)`
: Drag the character before the cursor forward over the character at the cursor, moving the cursor forward as well. If the insertion point is at the end of the line, then this transposes the last two characters of the line. 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 the current (or following) word. With a negative argument, uppercase the previous word, but do not move the cursor.

`downcase-word (M-l)`
: Lowercase the current (or following) word. With a negative argument, lowercase the previous word, but do not move the cursor.

`capitalize-word (M-c)`
: Capitalize the current (or following) word. With a negative argument, capitalize the previous word, but do not move the cursor.

21.4.4 Killing And Yanking

`kill-line (C-k)`
: Kill the text from the current cursor position to the end of the line.

`backward-kill-line (C-x Rubout)`
: Kill backward to the beginning of the line.

`unix-line-discard (C-u)`
: Kill backward from the cursor to the beginning of the current line. The killed text is saved on the kill-ring.

`kill-whole-line ()`
: Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word (M-d)`
: Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word (M-DEL)`
: Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`
: Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`
: Delete all spaces and tabs around point. By default, this is unbound.

`kill-region ()`
: Kill the text between the point and the *mark* (saved cursor position). This text is referred to as the *region*. By default, this command is unbound.

`copy-region-as-kill ()`
: Copy the text in the region to the kill buffer, so it can be yanked right away. By default, this command is unbound.

`copy-backward-word ()`
: Copy the word before point to the kill buffer. The word boundaries are the same as `backward-word`. By default, this command is unbound.

`copy-forward-word ()`
: Copy the word following point to the kill buffer. The word boundaries are the same as `forward-word`. By default, this command is unbound.

`yank (C-y)`
: Yank the top of the kill ring into the buffer at the current cursor position.

`yank-pop (M-y)`
: Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

21.4.5 Specifying Numeric Arguments

`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 ()`
: This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing `universal-argument` again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit

or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on. By default, this is not bound to a key.

21.4.6 Letting Readline Type For You

`complete (TAB)`
: Attempt to do completion on the text before the cursor. This is application-specific. 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, and so on.

`possible-completions (M-?)`
: List the possible completions of the text before the cursor.

`insert-completions (M-*)`
: Insert all completions of the text before point that would have been generated by `possible-completions`.

`menu-complete ()`
: Similar to `complete`, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of `menu-complete` steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung and the original text is restored. An argument of *n* moves *n* positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to `TAB`, but is unbound by default.

`delete-char-or-list ()`
: Deletes the character under the cursor if not at the beginning or end of the line (like `delete-char`). If at the end of the line, behaves identically to `possible-completions`. This command is unbound by default.

21.4.7 Keyboard Macros

`start-kbd-macro (C-x ()`
: Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro (C-x ))`
: Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro (C-x e)`
: Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

21.4.8 Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`
: Read in the contents of the inputrc file, and incorporate any bindings or variable assignments found there.

`abort (C-g)`
: Abort the current editing command and ring the terminal's bell (subject to the setting of `bell-style`).

`do-uppercase-version (M-a, M-b, M-x, ...)`
: If the metafied character *x* is lowercase, run the command that is bound to the corresponding uppercase character.

`prefix-meta (ESC)`
: Make the next character typed be metafied. This is for keyboards without a meta key. Typing '`ESC f`' is equivalent to typing '`M-f`'.

`undo (C-_, C-x C-u)`
: Incremental undo, separately remembered for each line.

`revert-line (M-r)`
: Undo all changes made to this line. This is like executing the `undo` command enough times to get back to the beginning.

`tilde-expand (M-~)`
: Perform tilde expansion on the current word.

`set-mark (C-@)`
: Set the mark to the current point. If a numeric argument is supplied, the mark is set to that position.

`exchange-point-and-mark (C-x C-x)`
: Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

`character-search (C-])`
: A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.

`character-search-backward (M-C-])`
: A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.

`insert-comment (M-#)`
: The value of the `comment-begin` variable is inserted at the beginning of the current line, and the line is accepted as if a newline had been typed.

`dump-functions ()`
: Print all of the functions and their key bindings to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`dump-variables ()`
: Print all of the settable variables and their values to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`dump-macros ()`
: Print all of the Readline key sequences bound to macros and the strings they ouput. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

21.5 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. The Readline `vi` mode behaves as specified in the POSIX 1003.2 standard.

In order to switch interactively between `emacs` and `vi` editing modes, use the command M-C-j (toggle-editing-mode). The Readline default is `emacs` mode.

When you enter a line in `vi` mode, you are already placed in 'insertion' mode, as if you had typed an '`i`'. Pressing ⟨ESC⟩ switches you into 'command' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with '`k`' and subsequent lines with '`j`', and so forth.

22 Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint. It should be considered a user's guide.

22.1 History Expansion

The History library provides a history expansion feature that is similar to the history expansion provided by `csh`. This section describes the syntax used to manipulate the history information.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion takes place in two parts. The first is to determine which line from the history list should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the history is called the *event*, and the portions of that line that are acted upon are called *words*. Various *modifiers* are available to manipulate the selected words. The line is broken into words in the same fashion that Bash does, so that several words surrounded by quotes are considered one word. History expansions are introduced by the appearance of the history expansion character, which is '`!`' by default.

22.1.1 Event Designators

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

`!` Start a history substitution, except when followed by a space, tab, the end of the line, '`=`' or '`(`'.

`!`*n* Refer to command line *n*.

`!-`*n* Refer to the command *n* lines back.

`!!` Refer to the previous command. This is a synonym for '`!-1`'.

`!`*string* Refer to the most recent command starting with *string*.

`!?`*string*`[?]`
: Refer to the most recent command containing *string*. The trailing '`?`' may be omitted if the *string* is followed immediately by a newline.

`^`*string1*`^`*string2*`^`
: Quick Substitution. Repeat the last command, replacing *string1* with *string2*. Equivalent to `!!:s/`*string1*`/`*string2*`/`.

`!#` The entire command line typed so far.

22.1.2 Word Designators

Word designators are used to select desired words from the event. A ':' separates the event specification from the word designator. It may 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 0 (zero). Words are inserted into the current line separated by single spaces.

`0 (zero)` The `0`th word. For many applications, this is the command word.

n The *n*th word.

`^` The first argument; that is, word 1.

`$` The last argument.

`%` The word matched by the most recent '?*string*?' search.

x-*y* A range of words; '-*y*' abbreviates '0-*y*'.

`*` All of the words, except the `0`th. This is a synonym for '1-$'. It is not an error to use '*' if there is just one word in the event; the empty string is returned in that case.

x`*` Abbreviates '*x*-$'

x`-` Abbreviates '*x*-$' like '*x**', but omits the last word.

If a word designator is supplied without an event specification, the previous command is used as the event.

22.1.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a ':'.

`h` Remove a trailing pathname component, leaving only the head.

`t` Remove all leading pathname components, leaving the tail.

`r` Remove a trailing suffix of the form '.*suffix*', leaving the basename.

`e` Remove all but the trailing suffix.

`p` Print the new command but do not execute it.

`s/`*old*`/`*new*`/`
Substitute *new* for the first occurrence of *old* in the event line. Any delimiter may be used in place of '/'. The delimiter may be quoted in *old* and *new* with a single backslash. If '&' appears in *new*, it is replaced by *old*. A single backslash will quote the '&'. The final delimiter is optional if it is the last character on the input line.

`&` Repeat the previous substitution.

`g` Cause changes to be applied over the entire event line. Used in conjunction with 's', as in `gs/`*old*`/`*new*`/`, or with '&'.

Appendix A Installing GDB

If you obtain GDB (HP WDB 2.0) as part of the HP ANSI C, HP ANSI C++ Developer's Kit for HP-UX Release 11.0, or HP Fortran, you do not have to take any special action to build or install GDB.

If you obtain GDB (HP WDB 2.0) from an HP web site, you may download either an `swinstall` package or a source tree, or both.

Most customers will want to install the GDB binary that is part of the `swinstall` package. To do so, use a command of the form

`/usr/sbin/swinstall -s` *package-name* `WDB`

Alternatively, it is possible to build GDB from the source distribution. If you who want to modify the debugger sources to tailor GDB to your needs you may wish to do this. The source distribution consists of a `tar` file containing the source tree rooted at '`gdb-4.17/...`'. The instructions that follow describe how to build a '`gdb`' executable from this source tree. HP believes that these instructions apply to the HP WDB 2.0 source tree that it distributes. However, HP does not explicitly support building a '`gdb`' for any non-HP platform from the HP WDB 2.0 source tree. It may work, but HP has not tested it for any platforms other than those described in the HP WDB 2.0 Release Notes.

You can find additional information specific to Hewlett-Packard in the '`README.HP.WDB`' file at the root of the source tree.

GDB comes with a `configure` script that automates the process of preparing GDB for installation; you can then use `make` to build the `gdb` program.[1]

The GDB distribution includes all the source code you need for GDB in a single directory, whose name is usually composed by appending the version number to '`gdb`'.

For example, the GDB version 19991101 distribution is in the '`gdb-19991101`' directory. That directory contains:

`gdb-19991101/configure` (and supporting files)
: script for configuring GDB and all its supporting libraries

`gdb-19991101/gdb`
: the source specific to GDB itself

`gdb-19991101/bfd`
: source for the Binary File Descriptor library

`gdb-19991101/include`
: GNU include files

`gdb-19991101/libiberty`
: source for the '`-liberty`' free software library

`gdb-19991101/opcodes`
: source for the library of opcode tables and disassemblers

`gdb-19991101/readline`
: source for the GNU command-line interface

[1] If you have a more recent version of GDB than 19991101, look at the '`README`' file in the sources; we may have improved the installation procedures since publishing this manual.

`gdb-19991101/glob`
: source for the GNU filename pattern-matching subroutine

`gdb-19991101/mmalloc`
: source for the GNU memory-mapped malloc package

The simplest way to configure and build GDB is to run `configure` from the '`gdb-`*version-number*' source directory, which in this example is the '`gdb-19991101`' directory.

First switch to the '`gdb-`*version-number*' source directory if you are not already in it; then run `configure`. Pass the identifier for the platform on which GDB will run as an argument.

For example:

```
cd gdb-19991101
./configure host
make
```

where *host* is an identifier such as '`sun4`' or '`decstation`', that identifies the platform where GDB will run. (You can often leave off *host*; `configure` tries to guess the correct value by examining your system.)

Running '`configure` *host*' and then running `make` builds the '`bfd`', '`readline`', '`mmalloc`', and '`libiberty`' libraries, then `gdb` itself. The configured source files, and the binaries, are left in the corresponding source directories.

`configure` is a Bourne-shell (`/bin/sh`) script; if your system does not recognize this automatically when you run a different shell, you may need to run `sh` on it explicitly:

```
sh configure host
```

If you run `configure` from a directory that contains source directories for multiple libraries or programs, such as the '`gdb-19991101`' source directory for version 19991101, `configure` creates configuration files for every directory level underneath (unless you tell it not to, with the '`--norecursion`' option).

You can run the `configure` script from any of the subordinate directories in the GDB distribution if you only want to configure that subdirectory, but be sure to specify a path to it.

For example, with version 19991101, type the following to configure only the `bfd` subdirectory:

```
cd gdb-19991101/bfd
../configure host
```

You can install `gdb` anywhere; it has no hardwired paths. However, you should make sure that the shell on your path (named by the '`SHELL`' environment variable) is publicly readable. Remember that GDB uses the shell to start your program—some systems refuse to let GDB debug child processes whose programs are not readable.

A.1 Compiling GDB in another directory

If you want to run GDB versions for several host or target machines, you need a different `gdb` compiled for each combination of host and target. `configure` is designed to make this easy by allowing you to generate each configuration in a separate subdirectory, rather than in the source directory. If your `make` program handles the '`VPATH`' feature (GNU `make` does), running `make` in each of these directories builds the `gdb` program specified there.

To build `gdb` in a separate directory, run `configure` with the '`--srcdir`' option to specify where to find the source. (You also need to specify a path to find `configure` itself from your working directory. If the path to `configure` would be the same as the argument to '`--srcdir`', you can leave out the '`--srcdir`' option; it is assumed.)

For example, with version 19991101, you can build GDB in a separate directory for a Sun 4 like this:

```
cd gdb-19991101
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-19991101/configure sun4
make
```

When `configure` builds a configuration using a remote source directory, it creates a tree for the binaries with the same structure (and using the same names) as the tree under the source directory. In the example, you'd find the Sun 4 library '`libiberty.a`' in the directory '`gdb-sun4/libiberty`', and GDB itself in '`gdb-sun4/gdb`'.

One popular reason to build several GDB configurations in separate directories is to configure GDB for cross-compiling (where GDB runs on one machine—the *host*—while debugging programs that run on another machine—the *target*). You specify a cross-debugging target by giving the '`--target=`*target*' option to `configure`.

When you run `make` to build a program or library, you must run it in a configured directory—whatever directory you were in when you called `configure` (or one of its subdirectories).

The `Makefile` that `configure` generates in each source directory also runs recursively. If you type `make` in a source directory such as '`gdb-19991101`' (or in a separate configured directory configured with '`--srcdir=`*dirname*`/gdb-19991101`'), you will build all the required libraries, and then build GDB.

When you have multiple hosts or targets configured in separate directories, you can run `make` on them in parallel (for example, if they are NFS-mounted on each of the hosts); they will not interfere with each other.

A.2 Specifying names for hosts and targets

The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, but some short predefined aliases are also supported. The full naming scheme encodes three pieces of information in the following pattern:

architecture-vendor-os

For example, you can use the alias `sun4` as a *host* argument, or as the value for *target* in a `--target=`*target* option. The equivalent full name is '`sparc-sun-sunos4`'.

The `configure` script accompanying GDB does not provide any query facility to list all supported host and target names or aliases. `configure` calls the Bourne shell script `config.sub` to map abbreviations to full names; you can read the script, if you wish, or you can use it to test your guesses on abbreviations—for example:

```
% sh config.sub i386-linux
i386-pc-linux-gnu
% sh config.sub alpha-linux
alpha-unknown-linux-gnu
% sh config.sub hp9k700
hppa1.1-hp-hpux
% sh config.sub sun4
sparc-sun-sunos4.1.1
% sh config.sub sun3
m68k-sun-sunos4.1.1
% sh config.sub i986v
Invalid configuration 'i986v': machine 'i986v' not recognized
```

`config.sub` is also distributed in the GDB source directory ('`gdb-19991101`', for version 19991101).

A.3 `configure` options

Here is a summary of the `configure` options and arguments that are most often useful for building GDB. `configure` also has several other options not listed here. See Info file '`configure.info`', node '`What Configure Does`', for a full explanation of `configure`.

```
configure [--help]
          [--prefix=dir]
          [--exec-prefix=dir]
          [--srcdir=dirname]
          [--norecursion] [--rm]
          [--target=target]
          host
```

You may introduce options with a single '`-`' rather than '`--`' if you prefer; but you may abbreviate option names if you use '`--`'.

`--help` Display a quick summary of how to invoke `configure`.

`--prefix=`*dir*
: Configure the source to install programs and files under directory '*dir*'.

`--exec-prefix=`*dir*
: Configure the source to install programs under directory '*dir*'.

`--srcdir=`*dirname*
: **Warning: using this option requires GNU `make`, or another `make` that implements the `VPATH` feature.**
Use this option to make configurations in directories separate from the GDB source directories. Among other things, you can use this to build (or maintain) several configurations simultaneously, in separate directories. `configure` writes configuration specific files in the current directory, but arranges for them to use the source in the directory *dirname*. `configure` creates directories under the working directory in parallel to the source directories below *dirname*.

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

`--target=`*target*
: Configure GDB for cross-debugging programs running on the specified *target*. Without this option, GDB is configured to debug programs that run on the same machine (*host*) as GDB itself.

  There is no convenient way to generate a list of all available targets.

host ...
: Configure GDB to run on the specified *host*.

  There is no convenient way to generate a list of all available hosts.

There are many other options available as well, but they are generally needed for special purposes only.

Index

C

G

H

I

J

K

L

M

N

O

T

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt,
cmb10 at 10.95pt, and
cmsl10 at 10.95pt
are used for emphasis.

Table of Contents