

**GNUPRO™ TOOLKIT**

# **GNUPro Debugging Tools**

---

98r2

**CYGNUS**

Copyright © 1991-1998 Cygnus.

All rights reserved.

GNUPro™, the GNUPro™ logo and the Cygnus logo are all trademarks of Cygnus.

All other brand and product names are trademarks of their respective owners.

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

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

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

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: **`doc@cygnus.com`**.

Part #: 300-400-1010043-98r2

---

## GNUPro warranty

The GNUPro Toolkit 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. This version of GNUPro Toolkit is supported for customers of Cygnus.

For non-customers, GNUPro Toolkit software has NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

## How to contact Cygnus

Use the following means to contact Cygnus.

### *Cygnus Headquarters*

1325 Chesapeake Terrace  
Sunnyvale, CA 94089 USA  
Telephone (toll free): +1 800 CYGNUS-1  
Telephone (main line): +1 408 542 9600  
Telephone (hotline): +1 408 542 9601  
FAX: +1-408 542 9699  
(Faxes are answered 8 a.m.–5 p.m., Monday through Friday.)  
email: [info@cygnus.com](mailto:info@cygnus.com)  
Website: [www.cygnus.com](http://www.cygnus.com).

### *Cygnus United Kingdom*

36 Cambridge Place  
Cambridge CB2 1NS  
United Kingdom  
Telephone: +44 1223 728728  
FAX: +44 1223 728728  
email: [info@cygnus.co.uk/](mailto:info@cygnus.co.uk/)

### *Cygnus Japan*

Nihon Cygnus Solutions  
Madre Matsuda Building  
4-13 Kioi-cho Chiyoda-ku  
Tokyo 102-0094  
Telephone: +81 3 3234 3896  
FAX: +81 3 3239 3300  
email: [info@cygnus.co.jp](mailto:info@cygnus.co.jp)  
Website: <http://www.cygnus.co.jp/>

Use the hotline (+1 408 542 9601) to get help, although the most reliable and most expedient means to resolve problems with GNUPro Toolkit is by using the Cygnus Web Support site:

<http://support.cygnus.com>

# Contents

---

GNUPro warranty .....	iii
How to contact Cygnus .....	iv
 <b>Debugging with GDB</b>	
<i>Summary of the GNU debugger, GDB</i> .....	3
GDB as free software .....	5
Contributors to GDB .....	6
<i>Installing GDB</i> .....	9
Compiling GDB in another directory .....	11
Specifying names for hosts and targets .....	12
configure options .....	13
<i>Getting in and out of GDB</i> .....	15
Invoking GDB .....	16
Choosing files .....	17
Choosing modes .....	19
Quitting GDB .....	20
Shell commands .....	21
<i>GDB commands</i> .....	23
Command syntax .....	24
Command completion .....	25
Getting help .....	27

<b><i>Running programs under GDB</i></b> .....	<b>29</b>
Compiling for debugging .....	30
Starting your program .....	31
Your program's arguments .....	33
Your program's environment .....	34
Your program's working directory .....	36
Your program's input and output .....	37
Debugging an already-running process.....	38
Killing the child process.....	39
Additional process information.....	40
Debugging programs with multiple threads.....	41
Debugging programs with multiple processes .....	43
<b><i>Stopping and continuing</i></b> .....	<b>45</b>
Breakpoints, watchpoints, and exceptions .....	47
Setting breakpoints.....	48
Setting watchpoints .....	52
Breakpoints and exceptions.....	53
Deleting breakpoints .....	54
Disabling breakpoints.....	55
Break conditions.....	57
Breakpoint command lists.....	59
Breakpoint menus.....	61
Continuing and stepping .....	62
Signals.....	65
Stopping and starting multi-thread programs.....	67
<b><i>Examining the stack</i></b> .....	<b>69</b>
Stack frames .....	70
Backtraces .....	71
Selecting a frame.....	72
Information about a frame.....	74
<b><i>Examining source files</i></b> .....	<b>77</b>
Printing source lines .....	78
Searching source files .....	80
Specifying source directories .....	81
Source and machine code.....	82
<b><i>Examining data</i></b> .....	<b>85</b>
Expressions .....	86
Program variables.....	87
Artificial arrays .....	89

Output formats .....	90
Examining memory .....	91
Automatic display .....	93
Print settings.....	95
Value history.....	100
Convenience variables .....	101
Registers .....	103
Floating point hardware .....	105
<b>Using GDB with different languages .....</b>	<b>107</b>
Switching between source languages.....	108
<i>List of filename extensions and languages .....</i>	<i>108</i>
<i>Setting the working language .....</i>	<i>109</i>
<i>Having GDB infer the source language .....</i>	<i>109</i>
Displaying the language.....	110
Type and range checking.....	111
<i>An overview of type checking.....</i>	<i>111</i>
<i>An overview of range checking.....</i>	<i>112</i>
Supported languages .....	114
<b>Examining the symbol table .....</b>	<b>127</b>
<b>Altering execution.....</b>	<b>131</b>
Assignment to variables .....	132
Continuing at a different address .....	133
Giving your program a signal .....	134
Returning from a function.....	135
Calling program functions.....	136
Patching programs.....	137
<b>GDB files .....</b>	<b>139</b>
Commands to specify files.....	140
Errors reading symbol files .....	145
<b>Specifying a debugging target .....</b>	<b>147</b>
Active targets .....	148
Commands for managing targets .....	149
Choosing target byte order .....	152
Remote debugging .....	153
The GDB remote serial protocol.....	154
<i>What the stub can do for you .....</i>	<i>155</i>
<i>What you must do for the stub .....</i>	<i>156</i>
<i>Putting it all together.....</i>	<i>158</i>
<i>Communication protocol .....</i>	<i>159</i>

<i>Using the <code>gdbserver</code> program</i>	160
<i>Using the <code>gdbserve.nlm</code> program</i>	162
<i>GDB with a remote i960 (Nindy)</i>	163
<i>The UDI protocol for AMD29K</i>	164
<i>GDB with a Tandem ST2000</i>	167
<i>GDB and VxWorks</i>	168
<i>GDB and SPARClet</i>	169
<i>Setting <code>file</code> to debug</i>	170
<i>Connecting to SPARClet</i>	170
<i>SPARClet download</i>	171
<i>GDB and Hitachi microprocessors</i>	171
<i>GDB and remote MIPS boards</i>	173
<b><i>Controlling GDB</i></b>	<b>177</b>
Prompt	178
Command editing	179
Command history	180
Screen size	182
Numbers	183
Optional warnings and messages	184
<b><i>Canned sequences of commands</i></b>	<b>185</b>
User-defined commands	186
User-defined command hooks	188
Command files	189
Commands for controlled output	190
<b><i>Using GDB under GNU Emacs</i></b>	<b>193</b>
Emacs considerations with GDB	194
Keystroke sequences for GDB with Emacs	196
<b><i>Reporting bugs in GDB</i></b>	<b>199</b>
Have you found a bug?	200
How to report bugs	201
<b><i>Command line editing</i></b>	<b>205</b>
Readline interaction	207
Readline bare essentials	208
Readline movement commands	209
Readline killing commands	210
Readline arguments	211
Readline init file	212
Variable settings for Readline	213
Key bindings for Readline	214



---

Commands for moving around in Readline .....	215
Commands for manipulating history with Readline .....	216
Commands for changing text in Readline .....	217
Killing and yanking .....	218
Specifying numeric arguments .....	219
Letting Readline type for you .....	220
<i>Some miscellaneous Readline commands</i> .....	221
Readline in vi mode .....	222
<b><i>Using history interactively</i></b> .....	223
History interaction .....	224
<i>Event designators</i> .....	224
<i>Word designators</i> .....	224
<i>Modifiers</i> .....	225
<b><i>Formatting documentation</i></b> .....	227
<b>Index</b> .....	229



# **Debugging with GDB**

---

Copyright © 1991-1998 Cygnus.

All rights reserved.

GNUPro™, the GNUPro™ logo and the Cygnus logo are all trademarks of Cygnus.

All other brand and product names are trademarks of their respective owners.

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

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

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

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: **`doc@cygnus.com`**.

# Summary of the GNU debugger, GDB

---

The purpose of a debugger such as the GNU debugger, GDB, is to allow you to see what is going on *inside* another program while it executes—or what another program was doing at the moment it crashed. GDB can do four main kinds of things to help you catch bugs.

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

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

- “Installing GDB” on page 9
- “Getting in and out of GDB” on page 15
- “GDB commands” on page 23
- “Running programs under GDB” on page 29
- “Stopping and continuing” on page 45
- “Examining the stack” on page 69
- “Examining source files” on page 77
- “Examining data” on page 85

- 
- “Using GDB with different languages” on page 107
  - “Examining the symbol table” on page 127
  - “Altering execution” on page 131
  - “GDB files” on page 139
  - “Specifying a debugging target” on page 147
  - “Controlling GDB” on page 177
  - “Canned sequences of commands” on page 185
  - “Using GDB under GNU Emacs” on page 193
  - “Reporting bugs in GDB” on page 199
  - “Command line editing” on page 205
  - “Using history interactively” on page 223
  - “Formatting documentation” on page 227

# GDB as free software

GDB is *free software*, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms. Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else. To see the GNU General Public License, see “General licenses and terms for using GNUPro Toolkit” in *Getting Started with GNUPro Toolkit*.

## 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.

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

So that they may not regard their long labor as thankless, we particularly thank those who shepherded GDB through major releases: Stan Shebs (release 4.14), Fred Fish (releases 4.13, 4.12, 4.11, 4.10, and 4.9), Stu Grossman and John Gilmore (releases 4.8, 4.7, 4.6, 4.5, and 4.4), John Gilmore (releases 4.3, 4.2, 4.1, 4.0, and 3.9); Jim Kingdon (releases 3.5, 3.4, and 3.3); and Randy Smith (releases 3.2, 3.1, and 3.0). As major maintainer of GDB for some period, each contributed significantly to the structure, stability, and capabilities of the entire debugger.

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

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

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

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

Adam de Boor and Bradley Davis contributed the ISI Optimum V support.

Per Bothner, Noboyuki Hikichi, and Alessandro Forin contributed MIPS support.

Jean-Daniel Fekete contributed Sun 386i support.

Chris Hanson improved the HP9000 support.

Noboyuki Hikichi and Tomoyuki Hasei contributed Sony/News OS 3 support.

David Johnson contributed Encore Umax support.

Jyrki Kuoppala contributed Altos 3068 support.



Jeff Law contributed HP PA and SOM support.

Keith Packard contributed NS32K support.

Doug Rabson contributed Acorn Risc Machine support.

Bob Rusk contributed Harris Nighthawk CXUX support.

Chris Smith contributed Convex support (and Fortran debugging).

Jonathan Stone contributed Pyramid support.

Michael Tiemann contributed SPARC support.

Tim Tucker contributed support for the Gould NP1 and Gould Pownode.

Pace Willison contributed Intel 386 support.

Jay Vosburgh contributed Symmetry support.

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

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

Patrick Duval, Ted Goldstein, Vikram Koka and Glenn Engel helped develop remote debugging.

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

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

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

Fred Fish wrote most of the support for Unix System V4. He also enhanced the command-completion support to cover C++ overloaded symbols.

Hitachi America, Ltd. sponsored the support for Hitachi microprocessors.

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

Stu Grossman wrote `gdbserver`.

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



# Installing GDB

---

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. The 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 the ‘`gdb`’ program. For example, the GDB version 4.17-98r2 distribution is in the ‘`gdb-4.17-98r2`’ directory.

That directory contains the following files.

- `gdb-4.17-98r2/configure` (and supporting files)  
Script for configuring GDB and all its supporting libraries
- `gdb-4.17-98r2/gdb`  
Source specific to GDB
- `gdb-4.17-98r2/bfd`  
Source for the Binary File Descriptor library
- `gdb-4.17-98r2/include`  
GNU include files
- `gdb-4.17-98r2/libiberty`  
Source for the ‘`-liberty`’ free software library
- `gdb-4.17-98r2/opcodes`  
Source for the library of opcode tables and disassemblers
- `gdb-4.17-98r2/readline`  
Source for the GNU command-line interface

- `gdb-4.17-98r2/glob`  
Source for the GNU filename pattern-matching subroutine
- `gdb-4.17-98r2/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 the following examples is designated as the ‘`gdb-4.17-98r1`’ 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.

Consider the following, for example.

```
cd gdb-4.17-98r2
./configure host
make
```

`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-4.17-98r2`’ source directory for version 4.17-98r1, `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 4.17-98r2, type the following to configure only the `bfd` subdirectory:

```
cd gdb-4.17-98r2/bfd
../configure host
```

You can install `gdb` anywhere; it has no hardwired paths. However, you should make sure that the shell on your path (named by the ‘`SHELL`’ environment variable) is publicly readable. Remember that GDB uses the shell to start your program—some systems refuse to let GDB debug child processes whose programs are not readable.

## Compiling GDB in another directory

If you want to run GDB versions for several host or target machines, you need a different `gdb` compiled for each combination of host and target. `configure` is designed to make this easy by allowing you to generate each configuration in a separate subdirectory, rather than in the source directory. If your `make` program handles the `'VPATH'` feature (GNU `make` does; for more on the `VPATH` option, see *Using make* in **GNUPro Utilities**), running `make` in each of these directories builds the `gdb` program specified there.

To build `gdb` in a separate directory, run `configure` with the `'--srcdir'` option to specify where to find the source. (You also need to specify a path to find `configure` itself from your working directory. If the path to `configure` would be the same as the argument to `'--srcdir'`, you can leave out the `'--srcdir'` option; it is assumed.) For example, with the current version, you can build GDB in a separate directory for your machine, using the following declaration (where *version* is the version which you have installed by default and *host* is the host machine with which you installed the tools).

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

When `configure` builds a configuration using a remote source directory, it creates a tree for the binaries with the same structure (and using the same names) as the tree under the source directory. In the example, you'd find the *host* library, `libiberty.a`, in the directory `'gdb-host/libiberty'`, and GDB itself in `'gdb-host/gdb'`.

One popular reason to build several GDB configurations in separate directories is to configure GDB for cross-compiling (where GDB runs on one machine—the *host*—while debugging programs that run on another machine—the *target*). You specify a cross-debugging target by giving the `'--target=target'` option to `configure`. When you run `make` to build a program or library, you must run it in a configured directory—whatever directory you were in when you called `configure` (or one of its subdirectories). The Makefile that `configure` generates in each source directory also runs recursively. If you type `make` in a source directory such as `'gdb-version'` (or in a separate directory configured with `'--srcdir=dirname/gdb-version'`), you will build all the required libraries, and then build GDB. When you have multiple hosts or targets configured in separate directories, you can run `make` on them in parallel (for example, if they are NFS-mounted on each of the hosts); they will not interfere with each other.

## Specifying names for hosts and targets

The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, but some short predefined aliases are also supported. The full naming scheme encodes three pieces of information in the following triplet pattern: *architecture-vendor-os*.

For example, use the alias, `sun4`, as a *host* argument, or as the value for target in a `--target=target` option.

'`sparc-sun-sunos4`' is the equivalent full name.

The `configure` script accompanying GDB does not provide any query facility to list all supported host and target names or aliases. `configure` calls the Bourne shell script, `config.sub`, to map abbreviations to full names; you can read the script, if you wish, or you can use it to test your guesses on abbreviations, as in the following example.

```
% sh config.sub sun4
sparc-sun-sunos4.1.1
% sh config.sub sun3
m68k-sun-sunos4.1.1
% sh config.sub decstation
mips-dec-ultrix4.2
% sh config.sub hp300bsd
m68k-hp-bsd
% sh config.sub i386v
i386-unknown-sysv
% sh config.sub i786v
Invalid configuration 'i786v': machine 'i786v' not recognized
```

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

## configure options

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

```
configure [--help]
[--prefix=dir]
[--srcdir=dirname]
[--norecursion][--rm]
[--target=target] host
```

You may introduce options with a single ‘-’ rather than ‘--’ if you prefer; but you may abbreviate option names if you use ‘--’.

`--help`

Display a quick summary of how to invoke `configure`.

`--prefix=dir`

Configure the source to install programs and files under directory ‘*dir*’.

`--srcdir=dirname`

Use this option to make configurations in directories separate from the GDB source directories. Among other things, you can use this to build (or maintain) several configurations simultaneously, in separate directories.

`configure` writes configuration specific files in the current directory, but arranges for them to use the source in the directory *dirname*.

`configure` creates directories under the working directory in parallel to the source directories below *dirname*.

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

`--norecursion`

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

`--rm`

Remove files otherwise built during configuration.

`--target=target`

Configure GDB for cross-debugging programs running on the specified *target*. Without this option, GDB is configured to debug programs that run on the same machine (*host*) as GDB itself. There is no convenient way to generate a list of all available targets.

*host...*

Configure GDB to run on the specified *host*. There is no convenient way to

generate a list of all available hosts.

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



## Getting in and out of GDB

---

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

The essentials are starting GDB and quitting GDB.

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

The following documentation discusses the main essentials of working with GDB.

- “Invoking GDB” on page 16
- “Choosing files” on page 17
- “Choosing modes” on page 19
- “Quitting GDB” on page 20
- “Shell commands” on page 21

## Invoking GDB

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

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

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

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

```
gdb program
```

You can also start with both an executable program and a core file specified as the following example's input and variables show, where the core file is signified as *core*.

```
gdb program core
```

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

```
gdb program 1234
```

Your machine hereby attaches GDB to process 1234 (unless you also have a file named '1234'; GDB does check for a core file first).

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

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

```
gdb -silent
```

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

To display all available options and briefly describe their use, use `gdb -help` as input (`'gdb -h'` is a shorter equivalent).

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

## Choosing files

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

Many options have both long and short forms; both are shown in 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 coredump format named *number*, in which case ‘-c’ specifies that file as a core dump to read).

`-command file`

`-x file`

Execute GDB commands from file, *file*. See “Command files” on page 189.

`-directory directory`

`-d directory`

Add *directory* to the path to search for source files.

`-m`

`-mapped`

If memory-mapped files are available on your system through the `mmap` system call, you can use this option to have GDB write the symbols from your program into a reusable file in the current directory. If the program you are debugging is

called `‘/tmp/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.

**WARNING:** This option depends on operating system facilities that are not supported on all systems.

`-r`

`-readnow`

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

The `-mapped` and `-readnow` options are typically combined in order to build a `‘.syms’` file that contains complete symbol information. (See “Commands to specify files” on page 140 for information.

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

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

## Choosing modes

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

`-nx`

`-n`

Do not execute commands from any initialization files (normally called `‘.gdbinit’`). Normally, the commands in these files are executed after all the command options and arguments have been processed. See “Command files” on page 189.

`-quiet`

`-q`

*Quiet.* Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

`-batch`

Run in batch mode. Exit with status 0 after processing all the command files specified with `‘-x’` (and all commands from initialization files, if not inhibited with `‘-n’`). Exit with non-zero status if an error occurs in executing the GDB commands in the command files. Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the following message does not issue when running in batch mode. Ordinarily, the message does issue whenever a program running under GDB control terminates.

**Program exited normally.**

`-cd directory`

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

`-fullname`

`-f`

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

`-b bps`

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

`-tty device`

Run using *device* for your program’s standard input and output.

## Quitting GDB

`quit`

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

If you have been using GDB to control an attached process or device, you can release it with the `detach` command (see “Debugging an already-running process” on page 38).

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

Otherwise GDB uses `/bin/sh`.

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

`make make-args`

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





# 4

## GDB commands

---

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

The following documentation discusses more GDB commands.

- “Command syntax” on page 24
- “Command completion” on page 25
- “Getting help” on page 27

## 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 (using the Return key just once) means to repeat the previous command. Certain commands (for example, `run`) will not repeat this way; such commands have unintentional repetition which might cause trouble and which it is unlikely you want to repeat.

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

GDB can also use Return in another way: to partition lengthy output, in a way similar to the common utility, `more` (see “Screen size” on page 182). Since it is easy to use Return one too many times in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files (see “Command files” on page 189).

## Command completion

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

Use the TAB key whenever you want GDB to fill out the rest of a word. If there is only one possibility, GDB fills in the word, and waits for you to finish the command (or use Return to enter it). For example, if you type `(gdb) info bre`, and use the TAB key, GDB fills in the rest of the word `'breakpoints'`, since that is the only `info` subcommand beginning with `'bre'`.

You can either use Return at this point, to run the `info breakpoints` command, or use the BACKSPACE key and enter something else, if `'breakpoints'` does not look like the command you expected. (If you were sure you wanted `info breakpoints` in the first place, you might as well just use Return immediately after `'info bre'`, to exploit command abbreviations rather than command completion). If there is more than one possibility for the next word when you use the TAB key, GDB sounds a bell. You can either supply more characters and try again, or just use the TAB key a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with `'make_'`, but when you type `b make_` and use the TAB key, GDB just sounds the bell. Using the TAB key again displays all the function names in your program that begin with those characters. For example, you type `(gdb) b make_` and then use the TAB key. GDB sounds the bell; you use the TAB key again, to see the following display.

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

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

**IMPORTANT:** `M-?` means using the META key (the diamond-marked key, or else, use ESC) and the `'?'` key as a command key sequence.

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

marks in GDB commands.

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

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

Use the `M-?` command key sequence this point.

```
bubble(double,double) bubble(int,int)
(gdb) b 'bubble(
```

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

```
(gdb) b bub
```

Use the `TAB` key at this point. GDB alters your input line then to the following declaration, and rings a bell.

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

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

## Getting help

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

`help`

`h`

You can use `help` (abbreviated ‘`h`’) with no arguments to display a short list of named classes of commands like the following output.

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

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

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

```
(gdb)
```

`help class`

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

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

```
List of commands:
```

```
show -- Generic command for showing things set with “set”
info -- Generic command for printing status
```

Type “`help`” followed by command name for full documentation.

Command name abbreviations are allowed if unambiguous.

```
(gdb)
```

`help command`

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

### `complete args`

The `complete args` command lists all the possible completions for the beginning of a command. With `args`, specify the beginning of the command you want completed; for example, `output` for `info`, `inspect` or `ignore`. This command is intentionally for use by GNU Emacs.

In addition to `help`, you can use the GDB commands `info` and `show` to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under `info` and under `show` in the Index point to all the subcommands. See “Index” on page 229 for specific commands.

### `info`

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

### `set`

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

### `show`

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

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

The following are three miscellaneous `show` subcommands, all of which are exceptional in lacking corresponding `set` commands.

### `show version`

Show what version of GDB is running. You should include this information in GDB bug reports. If multiple versions of GDB are in use at your site, you may occasionally want to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. The version number is also announced when you start GDB.

### `show copying`

Display information about permission for copying GDB.

### `show warranty`

Display the GNU “NO WARRANTY” statement.

# Running programs under GDB

---

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

For more discussion, see the following topics.

- “Compiling for debugging” on page 30
- “Starting your program” on page 31
- “Your program's arguments” on page 33
- “Your program's environment” on page 34
- “Your program's working directory” on page 36
- “Your program's input and output” on page 37
- “Debugging an already-running process” on page 38
- “Killing the child process” on page 39
- “Additional process information” on page 40
- “Debugging programs with multiple threads” on page 41
- “Debugging programs with multiple processes” on page 43

## Compiling for debugging

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

To request debugging information, specify the ‘-g’ option when you run the compiler.

Many C compilers are unable to handle the ‘-g’ and ‘-o’ options together. Using those compilers, you cannot generate optimized executables containing debugging information.

GCC, the GNU C compiler, supports ‘-g’ with or without ‘-o’, making it possible to debug optimized code. We recommend that you always use ‘-g’ whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

When you debug a program compiled with ‘-g -o’, remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable—because the compiler optimizes it out of existence.

Some things do not work as well with ‘-g -o’ as with just ‘-g’, particularly on machines with instruction scheduling. If in doubt, recompile with ‘-g’ alone, and if this fixes the problem, please report it to us as a bug (including a test case!).

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.



## 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 “Getting in and out of GDB” on page 15), or using the `file` or `exec-file` command (see “Commands to specify files” on page 140).

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

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do *before* starting your program. (You can change it after starting your program, but such changes only affect your program the next time you start it.) This information may be divided into the following four categories.

- *Arguments*

Specify the arguments to give your program as the arguments of the `run` command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the `SHELL` environment variable. See “Your program’s arguments” on page 33.

- *Environment*

Your program normally inherits its environment from GDB, but you can use the GDB commands `set environment` and `unset environment` to change parts of the environment that affect your program. See “Your program’s environment” on page 34.

- *Working directory*

Your program inherits its working directory from GDB. You can set the GDB working directory with the `cd` command in GDB. See “Your program’s working directory” on page 36.

- *Standard input and output*

Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the `run` command line, or you can use the `tty` command to set a different device for your program. See “Your program’s input and output” on page 37.

**WARNING:** While input and output redirection work, you cannot use pipes to pass the

output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

When you issue the `run` command, your program begins to execute immediately. See “Stopping and continuing” on page 45 for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. See “Examining data” on page 85.

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

## Your program's arguments

The arguments to your program can be specified by the arguments of the `run` command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. Your `SHELL` environment variable (if it exists) specifies what shell GDB uses. If you do not define `SHELL`, GDB uses `/bin/sh`.

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

`set args`

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

`show args`

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

## Your program's environment

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

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

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

`path directory`

Add *directory* to the front of the `PATH` environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by `:` or a whitespace. If *directory* is already in the path, it is moved to the front, so it is searched sooner.

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

`show paths`

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

`show environment [varname]`

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

`set environment varname[=] value`

Set environment variable, *varname*, to *value*. The value changes for your program only, not for GDB itself. *value* may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The *value* parameter is optional; if it is eliminated, the variable is set to a null value. For example, the command, `set env USER = foo`, tells a Unix program, when run, that its user is named `'foo'`. (The spaces around `'='` are used for clarity here; they are not actually required.)

`unset environment varname`

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

**WARNING:** GDB runs your program using the shell indicated by your `SHELL` environment

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

## Your program's working directory

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

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See “Commands to specify files” on page 140.

`cd` *directory*

Set the GDB working directory to *directory*.

`pwd`

Print the GDB working directory.

## Your program's input and output

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

`info terminal`

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

You can redirect your program's input and/or output using shell redirection with the `run` command. For example, `run > outfile` starts your program, diverting its output to the file 'outfile'. Another way to specify where your program should do input and output is with the `tty` command. This command accepts a file name as argument, and causes this file to be the default for future `run` commands.

It also resets the controlling terminal for the child process, for future `run` commands. For example, `tty /dev/ttyb` directs that processes started with subsequent `run` commands default to do input and output on the terminal '/dev/ttyb' and have that as their controlling terminal.

An explicit redirection in `run` overrides the `tty` command's effect on the input/output device, but not its effect on the controlling terminal.

When you use the `tty` command or redirect input in the `run` command, only the input for your program is affected. The input for GDB still comes from your terminal.

## Debugging an already-running process

`attach process-id`

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

`attach` does not repeat if you use the Return key a second time after executing the command.

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

When using `attach`, you should first use the `file` command to specify the program running in the process and load its symbol table. See “Commands to specify files” on page 140.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with `run`. You can insert breakpoints; you can `step` and `continue`; you can modify storage. If you would rather the process continue running, you may use the `continue` command after attaching GDB to the process.

`detach`

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

If you exit GDB or use the `run` command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see “Optional warnings and messages” on page 184).



## Killing the child process

`kill`

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

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump files while your program is running.

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

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

## Additional process information

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

`info proc`

Summarize available information about the process.

`info proc mappings`

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

`info proc times`

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

`info proc id`

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

`info proc status`

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

`info proc all`

Show all the above information about the process.

## Debugging programs with multiple threads

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

GDB provides the following facilities for debugging multi-thread programs.

- automatic notification of new threads
- ‘thread *threadno*’, a command to switch among threads
- ‘info threads’, a command to inquire about existing threads
- ‘thread apply [*threadno*][*all*] *args*’, a command to apply a command to a list of threads
- thread-specific breakpoints

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

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

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

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

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

In contrast, on an SGI system, the *systag* is simply something like ‘process 368’, with no further qualifier.

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

`info threads`

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

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

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

```
(gdb) info threads
3 process 35 thread 27 0x34e5 in sigpause ()
2 process 35 thread 23 0x34e5 in sigpause ()
* 1 process 35 thread 13 main (argc=1, argv=0x7fffffff8)
  at threadtest.c:68
```

`thread threadno`

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

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

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

`thread apply [threadno][all] args`

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

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

See "Stopping and starting multi-thread programs" on page 67 for more information about how GDB behaves when you stop and start programs with multiple threads.

See "Setting watchpoints" on page 52 for information about watchpoints in programs with multiple threads.

## Debugging programs with multiple processes

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

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call to `sleep` in the code which the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the `ps` program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see `attach` with "Debugging an already-running process" on page 38). From that point on you can debug the child process just like any other process to which you attached.



# Stopping and continuing

---

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and determine causes.

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. The following documentation discusses these topics.

- “Breakpoints, watchpoints, and exceptions” on page 47
- “Setting breakpoints” on page 48
- “Setting watchpoints” on page 52
- “Breakpoints and exceptions” on page 53
- “Deleting breakpoints” on page 54
- “Disabling breakpoints” on page 55
- “Break conditions” on page 57
- “Breakpoint command lists” on page 59
- “Breakpoint menus” on page 61
- “Continuing and stepping” on page 62

- “Signals” on page 65
- “Stopping and starting multi-thread programs” on page 67

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



## Breakpoints, watchpoints, and exceptions

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break` command and its variants (see “Setting breakpoints” on page 48) to specify the place where your program should stop by line number, function name or exact address in the program.

In languages with exception handling (such as GNU C++), you can also set breakpoints where an exception is raised (see “Breakpoints and exceptions” on page 53).

In SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can set breakpoints in shared libraries before the executable is run.

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see “Setting watchpoints” on page 52), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See “Automatic display” on page 93.

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

## Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable `'$bpnum'` records the number of the breakpoints you've set most recently; see "Convenience variables" on page 101 for a discussion of what you can do with convenience variables.

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

`break function`

Set a breakpoint at entry to function, *function*. When using source languages that permit overloading of symbols, such as C++, *function* may refer to more than one possible place to break. See "Breakpoint menus" on page 61 for a discussion of that situation.

`break +offset`

`break -offset`

Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

`break linenum`

Set a breakpoint in the current source file at line, *linenum*. That file is the last file whose source text was printed. This breakpoint stops your program just before it executes any of the code on that line.

`break filename:linenum`

Set a breakpoint at line, *linenum*, in source file, *filename*.

`break filename:function`

Set a breakpoint at entry to function, *function*, found in file, *filename*.

Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

`break *address`

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

`break`

When called without any arguments, `break` sets a breakpoint at the next instruction to be executed in the selected stack frame (see "Examining the stack" on page 69). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame.

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

executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint.

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

`break...if cond`

Set a breakpoint with condition, *cond*; evaluate the expression, *cond*, each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond*, evaluates as true. ‘...’ stands for one of the possible arguments described previously (or no argument) specifying where to break. See “Break conditions” on page 57 for more information on breakpoint conditions.

`tbreak args`

Set a breakpoint enabled only for one stop. *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See “Deleting breakpoints” on page 54.

`hbreak args`

Set a hardware-assisted breakpoint. *args* are the same as for the `break` command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClite DSU. DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can only take two data breakpoints, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones. See “Break conditions” on page 57.

`thbreak args`

Set a hardware-assisted breakpoint enabled only for one stop. *args* are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support and some target hardware may not have this support. See “Disabling breakpoints” on page 55 and “Break conditions” on page 57.

`rbreak regex`

Set breakpoints on all functions matching regular expression, *regex*. Sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the `break` command. You can delete them, disable them, or make them

conditional the same way as any other breakpoint. When debugging C++ programs, `rbreak` is useful for setting breakpoints on overloaded functions that are not members of any special classes.

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

Print a table of all breakpoints and watchpoints set and not deleted, with the following place-settings for each breakpoint.

### *Breakpoint Numbers*

#### *Type*

Breakpoint or watchpoint.

#### *Disposition*

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

#### *Enabled or Disabled*

Enabled breakpoints are marked with ‘y’. ‘n’ marks breakpoints that are not enabled.

#### *Address*

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

#### *What*

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

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

`info break` with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see “Examining memory” on page 91).

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

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see “Break conditions” on page Break conditions). 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, as in the following clarifications.

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

## Setting watchpoints

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

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

`watch expr`

Set a watchpoint for an expression. GDB will break when *expr* is written into by the program and its value changes. This can be used with the new trap-generation provided by SPARClite DSU. DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. For the data addresses, DSU facilitates the `watch` command. However the hardware breakpoint registers can only take two data watchpoints, and both watchpoints must be the same kind. For example, you can set two watchpoints with `watch` commands, two with `rwatch` commands, or two with `awatch` commands, but you cannot set one watchpoint with one command and the other with a different command. {No value for "GBDN"} will reject the command if you try to mix watchpoints. Delete or disable unused watchpoint commands before setting new ones.

`rwatch expr`

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

`awatch expr`

Set a watchpoint that will break when *args* is read and written into by the program. If you use both watchpoints, both must be set with the `awatch` command.

`info watchpoints`

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

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

## Breakpoints and exceptions

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

`catch exceptions`

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

You can use `info catch` to list active exception handlers. See “Information about a frame” on page 74. There are currently some limitations to exception handling in GDB.

- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits.
- You cannot raise an exception interactively.
- You cannot install an exception handler interactively.

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

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

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

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__raise_exception` (see “Breakpoints, watchpoints, and exceptions” on page 47).

With a conditional breakpoint (see “Break conditions” on page 57) that depends on the value of `id`, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

## Deleting breakpoints

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

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

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

`clear`

Delete any breakpoints at the next instruction to be executed in the selected stack frame (see “Selecting a frame” on page 72). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

`clear function`

`clear filename: function`

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

`clear linenum`

`clear filename: linenum`

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

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

Delete the breakpoints or watchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have `set confirm off`). You can abbreviate this command as `d`.



## Disabling breakpoints

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

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

A breakpoint or watchpoint can have four different states of enablement.

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

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

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

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

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

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

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

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

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

Enable the specified breakpoints to work once, then die. GDB deletes any of these breakpoints as soon as your program stops there.

Except for a breakpoint set with `tbreak` (see “Setting breakpoints” on page Setting breakpoints), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the previously listed commands. (The command, `until`, can set and delete a breakpoint of its own, but it doesn’t change the state of other breakpoints; see “Continuing and stepping” on page “Continuing and stepping” on page 62.)

## Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language (see “Expressions” on page 86). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

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

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

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see “Breakpoint command lists” on page 59).

Break conditions can be specified when a breakpoint is set, by using ‘if’ in the arguments to the *break* command. See “Setting breakpoints” on page 48 for more discussion. They can also be changed at any time with the *condition* command. The *watch* command does not recognize the *if* keyword; *condition* is the only way to impose a further condition on a watchpoint.

*condition bnum expression*

Specify *expression* as the break condition for breakpoint or watchpoint number, *bnum*. After you set a condition, breakpoint *bnum* stops your program only if the value of *expression* is true (nonzero, in C). When you use *condition*, GDB checks *expression* immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. GDB does not actually evaluate *expression* at the time the condition command is given, however. See “Expressions” on page 86.

*condition bnum*

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

unconditional breakpoint.

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

`ignore bnum count`

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

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

When you use `continue` to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to `continue`, rather than using `ignore`. See “Continuing and stepping” on page 62.

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.

Achieve the effect of the ignore count with a condition such as `'$foo-- < 0'` that uses a debugger convenience variable that is decremented each time. See “Convenience variables” on page 101.

## Breakpoint command lists

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

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

Specify a list of commands for breakpoint number, *bnum*. The commands themselves appear on the following lines. Type a line containing just *end* to terminate the commands. To remove all commands from a breakpoint, type *commands* and follow it immediately with *end*; that is, give no commands. With no *bnum* argument, *commands* refers to the last breakpoint or watchpoint set (not to the breakpoint most recently encountered).

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

You can use breakpoint commands to start your program up again. Simply use the *continue* command, or *step*, or any other command that resumes execution. Any other commands in the command list are ignored, after a command that resumes execution. This is because any time you resume execution (even with a simple *next* or *step*), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

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

The commands *echo*, *output*, and *printf* allow you to print precisely controlled output, and are often useful in silent breakpoints.

See “Commands for controlled output” on page 190.

For example, the following example shows how to use breakpoint commands to print the value of *x* at entry to *foo* whenever *x* is positive.

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

One application for breakpoint commands is to compensate for one bug so you can

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

The following is an example.

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

## Breakpoint menus

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

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

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] 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)
```

## Continuing and stepping

*Continuing* means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more “step” of your program, where “step” may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If due to a signal, you may want to use `handle`, or use ‘`signal 0`’ to resume execution. See “Signals” on page 65.)

`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 “Break conditions” on page 57).

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

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

To resume execution at a different place, you can use `return` (see “Returning from a function” on page 135) to go back to the calling function; or `jump` (see “Continuing at a different address” on page 133) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see “Breakpoints, watchpoints, and exceptions” on page 47 for more discussion) at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

`step`

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

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

The `step` command now only stops at the first instruction of a source line. This



prevents the multiple stops that used to occur in switch statements, for loops, etc. `step` continues to stop if a function that has debugging information is called within the line.

Also, the `step` command now only enters a subroutine if there is line number information for the subroutine. Otherwise it acts like the `next` command. This avoids problems when using `cc -g1` on MIPS machines. Previously, `step` entered subroutines if there was any debugging information about the routine.

`step count`

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

`next [count]`

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

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

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

`finish`

Continue running until just after function in the selected stack frame returns. Print the returned value (if any). Contrast this with the `return` command (see “Returning from a function” on page 135).

`u`

`until`

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

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

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

`until` may produce somewhat counter-intuitive results if the order of machine code does not match the order of the source lines. For instance, in the following example from a debugging session, the `f` (`frame`) command shows that

execution is stopped at line 206; yet when we use `until`, we get to line 195:

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

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop—even though the test in a C `for`-loop is written before the body of the loop.

The `until` command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement—not in terms of the actual machine code.

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

`until location`

`u location`

Continue running your program until either the specified location is reached, or the current stack frame returns. *location* is any of the forms of argument acceptable to `break` (see “Setting breakpoints” on page 48).

This form of the command uses breakpoints, and hence is quicker than `until` without an argument.

`stepi`

`si`

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

It is often useful to use ‘`display/i $pc`’ when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See “Automatic display” on page 93.

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

`nexti`

`ni`

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

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

## Signals

A *signal* is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in Unix `SIGINT` is the signal a program gets when you use an interrupt (often **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* (kill your program immediately) if the program has not specified in advance some other way to handle the signal. `SIGINT` does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

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

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

`info signals`

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

`info handle` is the new alias for `info signals`.

`handle signal keywords...`

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

The keywords allowed by the `handle` command can be abbreviated. Their full names use the following functionality.

`nostop`

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

`stop`

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

`print`

GDB should print a message when this signal happens.

`noprint`

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

`pass`

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

`nopass`

GDB should not allow your program to see this signal.

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

You can also use the `signal` command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with `'signal 0'`. See “Giving your program a signal” on page 134.

## Stopping and starting multi-thread programs

When your program has multiple threads (see “Debugging programs with multiple threads” on page 41), you can choose whether to set breakpoints on all threads, or on a particular thread.

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

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

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

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

You can use the *thread* qualifier on conditional breakpoints as well; in this case, place ‘thread threadno’ before the breakpoint condition, like the following example shows.

```
(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.



# Examining the stack

---

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

- “Stack frames” on page 70  
Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*. When your program stops, the GDB commands for examining the stack allow you to see all of this information. See also “Backtraces” on page 71.
- “Selecting a frame” on page 72  
One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select a particular frame.
- “Information about a frame” on page 74  
When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command.

## Stack frames

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

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

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

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward.

These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

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

`frame args`

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



## Backtraces

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

`backtrace`

`bt`

Print a backtrace of the entire stack: one line per frame for all frames in the stack. You can stop the backtrace at any time by using the system interrupt character sequence, `Ctrl-c`.

`backtrace n`

`bt n`

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

`backtrace -n`

`bt -n`

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

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

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

It was made with the command ‘`bt 3`’, showing the innermost three frames.

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

More stack frames would follow.

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

## Selecting a frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. The following commands are for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

`frame n`

`f n`

Select frame number *n*. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on.

The highest-numbered frame is the one for `main`.

`frame addr`

`f addr`

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

On the SPARC architecture, `frame` needs two addresses to select an arbitrary frame: a frame pointer and a stack pointer.

On the MIPS and Alpha architecture, it needs two addresses: a stack pointer and a program counter.

On the 29k architecture, `frame` needs three addresses: a register stack pointer, a program counter, and a memory stack pointer.

`up n`

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

`down n`

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

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

```
(gdb) up
```

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

```
10          read_input_file (argv[i]);
```

---

After such a printout, the `list` command with no arguments prints ten lines centered on the point of execution in the frame. See “Printing source lines” on page 78.

`up-silently n`

`down-silently n`

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

## Information about a frame

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

`frame`

`f`

When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated `f`. With an argument, this command is used to select a stack frame. See “Selecting a frame” on page 72.

`info frame`

`info f`

This command prints a verbose description of the selected stack frame, including the following information.

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

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

`info frame addr`

`info f addr`

Print a verbose description of the frame at address `addr`, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the `frame` command. See “Selecting a frame” on page 72.

`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`, to see the

update. See “Breakpoints and exceptions” on page 53.

## MIPS machines and the function stack

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

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

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

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

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

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

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

Display the current limit.



# Examining source files

---

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

- “Printing source lines” on page 78
- “Searching source files” on page 80
- “Specifying source directories” on page 81
- “Source and machine code” on page 82

When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see “Selecting a frame” on page 72), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source; see “Using GDB under GNU Emacs” on page 193 for details of using Emacs with GDB.

## Printing source lines

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

`list linenum`

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

`list function`

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

`list`

Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see “Examining the stack” on page 69), this prints lines centered around that line.

`list -`

Print lines just before the lines last printed.

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

`set listsize count`

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

`show listsize`

Display the number of lines that list prints.

Repeating a `list` command using the Return key discards the argument, so it is equivalent to typing `list`. This is more useful than listing the same lines again. An exception is made for an argument of ‘-’; 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. The following 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 specify source lines.

`list ,last`

Print lines ending with *last*.



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

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

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

`list`  
 As described for `list` on page list Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see “Examining the stack” on page 69), this prints lines centered around that line..

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

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

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

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

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

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

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

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

## Searching source files

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

`forward-search regex`

`search regex`

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

`reverse-search regex`

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

## Specifying source directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name.

**NOTE:** The executable search path is *not* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

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

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

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

```
directory dirname ...
```

```
dir dirname ...
```

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

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.

## Source and machine code

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

`info line linespec`

Print the starting and ending addresses of the compiled code for source line `linespec`. Specify source lines in any of the ways understood by the `list` command (see “Printing source lines” on page 78).

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

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

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

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

After `info line`, the default address for the `x` command is changed to the starting address of the line, so that `'x/i'` is sufficient to begin examining the machine code (see “Examining memory” on page 91). Also, this address is saved as the value of the convenience variable, `$_` (see “Convenience variables” on page 101).

`disassemble`

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

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

```
(gdb) disas 0x63e4 0x6404
Dump of assembler code from 0x63e4 to 0x6404:
0x63e4 <builtin_init+5340>:    ble 0x63f8 <builtin_init+5360>
0x63e8 <builtin_init+5344>:    sethi %hi(0x4c00), %o0
0x63ec <builtin_init+5348>:    ld [%i1+4], %o0
0x63f0 <builtin_init+5352>:    0x63fc <builtin_init+5364>
0x63f4 <builtin_init+5356>:    ld [%o0+4], %o0
0x63f8 <builtin_init+5360>:    or %o0, 0x1a4, %o0
0x63fc <builtin_init+5364>:    call 0x9288 <path_search>
```

---

```
0x6400 <builtin_init+5368>:    nop
End of assembler dump.
```

```
set assembly-language instruction-set
```

This command selects the instruction set to use when disassembling the program via the `disassemble` or `x/i` com-mands. It is useful for architectures that have more than one native instruction set. Currently, it is only defined for the Intel `x86` family. You can set `instruction-set` to either `i386` or `i8086`. The default is `i386`.



## 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 “Using GDB with different languages” on page 107).

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

`exp` is an expression (in the source language). By default the value of `exp` is printed in a format appropriate to its data type; you can choose a different format by specifying `‘/f’`, where `f` is a letter specifying the format; see “Output formats” on page 90.

```
print
print /f
```

If you omit `exp`, GDB displays the last value again (from the *value history*; see “Value history” on page 100). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See “Examining memory” on page 91. If you are interested in information about types, or about how the fields of a struct or class are declared, use the `ptype exp` command rather than `print`. See “Examining the symbol table” on page 127.

## Expressions

`print` and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is valid in an expression in GDB. This includes conditional expressions, function calls, casts and string constants. It unfortunately does not include symbols defined by preprocessor `#define` commands. GDB now supports array constants in expressions input by the user. *element*, *element ...* is the syntax to use. For example, you can now use the command, `print {1 2 3}` to build up an array in memory that is memory allocated in the target program.

**NOTE:** Because C is so widespread, most of the expressions shown in examples in this documentation are in C. See “Using GDB with different languages” on page 107 for information on how to use expressions in other languages.

In the following documentation, we discuss operators that you can use in GDB expressions regardless of your programming language. See also the introduction to “Examining data” on page 85.

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

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

@

‘@’ is a binary operator for treating parts of memory as arrays. See “Artificial arrays” on page 89 for more information.

::

‘::’ allows you to specify a variable in terms of the file or function where it is defined. See “Program variables” on page 87 for more information.

{ *type* } *addr*

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



## Program variables

The most common kind of expression to use is the name of a variable in your program. Variables in expressions are understood in the selected stack frame (see “Selecting a frame” on page “Selecting a frame” on page 72); they must be either *global* (sometimes referred to as *static*) or they must be *visible* (according to the scope rules of the programming language from the point of execution in that frame). Consider the following function example.

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

This means that you can examine and use the variable, `a`, whenever your program is executing within the function, `foo`; however, you can only use or examine the variable, `b`, while your program is executing inside the block where `b` is declared. There is an exception: you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file, using the colon-colon notation (`::`) as in the following example.

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

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

## Artificial arrays

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

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

If a program says `int *array = (int *) malloc (len * sizeof (int));`, you can print the contents of `array` with `p *array@len`.

The left operand of '@' must reside in memory. Array values made with '@' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see “Value history” on page Value history), 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)'` as the following example shows.

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

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see “Convenience variables” on page 101) as a counter in an expression that prints the first interesting value, and then repeat that expression using `Return`. For instance, suppose you have an array, `dtab`, of pointers to structures, and you are interested in the values of a field, `fv`, in each structure. The following is an example of what you might input, after which use the `Return` key twice.

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

## Output formats

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

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the print command with a slash and a format letter. 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”.
- a  
    Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:  

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```
- c  
    Regard as an integer and print it as a character constant.
- f  
    Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see “Registers” on page 103), type `p/x $pc`. No space is required before the slash because command names in GDB cannot contain a slash. To reprint the last value in the value history with a different format, you can use the `print` command with just a format and no expression. For example, `p/x` reprints the last value in hex.

---

\* ‘b’ cannot be used because these format letters are also used with the `x` command, where ‘b’ stands for “byte”; see “Examining memory” on page 91.

## Examining memory

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

```
x/ nfu addr
x addr
x
```

Use the `x` command to examine memory.

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

`n`, the *repeat count*

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

`f`, the *display format*

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

`u`, the *unit size*

The unit size uses any of the following variables.

- `b`  
Bytes.
- `h`  
Halfwords (two bytes).
- `w`  
Words (four bytes). This is the initial default.
- `g`  
Giant words (eight bytes).

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

`addr`, starting display address

`addr` is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. See “Expressions” on page 86 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 “Registers” on page 103) 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 “Source and machine code” on page 82.

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

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

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

## Automatic display

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

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

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

`display exp`

Add the expression, *exp*, to the list of expressions to display each time your program stops. See “Expressions” on page 86.

`display` does not repeat if you press Return again after using it.

`display/fmt exp`

For *fmt* specifying only a display format and not a size or count, add the expression *exp* to the auto-display list but arrange to display it each time in the specified format, *fmt*. See “Output formats” on page 90.

`display/fmt addr`

For *fmt* `'i'` or `'s'`, or including a unit-size or a number of units, add the expression, *addr*, as a memory address to be examined each time your program stops. Examining means in effect doing `'x/fmt addr'`. See “Examining memory” on page 91. For example, `'display/i $pc'` can be helpful, to see the machine instruction about to be executed each time execution stops (`'$pc'` is a common name for the program counter; see “Registers” on page 103).

`undisplay dnums ...`

`delete display dnums ...`

Remove item numbers *dnums* from the list of expressions to display. `undisplay` does not repeat if you use Return after using it. (Otherwise you would just get the error, ‘No display number...’.)

`disable display dnums ...`

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

`enable display dnums ...`

Enable display of item numbers, *dnums*. It becomes effective once again in auto

display of its expression, until you specify otherwise.

`display`

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

`info display`

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

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



## Print settings

GDB provides the following ways to control how *arrays*, *structures*, and *symbols* are printed. These settings are useful for debugging programs in any language:

```
set print address
set print address on
```

GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is `on`.

For example, the following is what a stack frame display looks like with `set print address on`:

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

```
set print address off
```

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

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

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

```
show print address
```

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset.

If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to clarify.

One way to do this is with `info line`, for example ‘`info line *0x4537`’.

Alternately, you can set GDB to print the source file and line number when it prints a symbolic address:

```
set print symbol-filename on
```

Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

```
set print symbol-filename off
```

Do not print source file name and line number of a symbol. This is the default.

```
show print symbol-filename
```

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

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

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

```
set print max-symbolic-offset max-offset
```

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

```
show print max-symbolic-offset
```

Ask how large the maximum offset is that GDB prints in a symbolic address.

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

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

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

Other settings control how different kinds of objects are printed:

```
set print array
```

```
set print array on
```

Pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

```
set print array off
```

Return to compressed format for arrays.

```
show print array
```

Show whether compressed or pretty format is selected for displaying arrays.

```
set print elements number-of-elements
```

Set a limit on how many elements of an array GDB will print. If GDB is printing a large array, it stops printing after it has printed the number of elements set by the

set print elements command. This limit also applies to the display of strings. Setting *number-of-elements* to zero means that the printing is unlimited.

show print elements

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

set print null-stop

Cause GDB to stop printing the characters of an array when the first `NULL` is encountered. This is useful when large arrays actually contain only short strings.

set print pretty on

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

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

set print pretty off

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

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

This is the default format.

show print pretty

Show which format GDB is using to print structures.

set print sevenbit-strings on

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

set print sevenbit-strings off

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

show print sevenbit-strings

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

set print union on

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

set print union off

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

```
show print union
```

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

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

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

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

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

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

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

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

The following settings are of interest when debugging C++ programs.

```
set print demangle
```

```
set print demangle on
```

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

```
show print demangle
```

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

```
set print asm-demangle
```

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

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

```
show print asm-demangle
```

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

```
set demangle-style style
```

Choose among several encoding schemes used by different compilers to represent C++ names. The choices for *style* are currently:

- auto

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

- `gnu`  
Decode based on the GNU C++ compiler (`g++`) encoding algorithm. This is the default.
- `lucid`  
Decode based on the Lucid C++ compiler (`lcc`) encoding algorithm.
- `arm`  
Decode using the algorithm in the *Annotated C++ Reference Manual* (Margaret A. Ellis & Bjarne Stroustrup, Addison Wesley, 1990).

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

- `foo`  
Show the list of formats.

`show demangle-style`

Display the encoding style currently in use for decoding C++ symbols.

`set print object`

`set print object on`

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

`set print object off`

Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

`show print object`

Show whether actual, or declared, object types are displayed.

`set print static-members`

`set print static-members on`

Print static members when displaying a C++ object. The default is on.

`set print static-members off`

Do not print static members when displaying a C++ object.

`show print static-members`

Show whether C++ static members are printed, or not.

`set print vtbl`

`set print vtbl on`

Pretty print C++ virtual function tables. The default is off.

`set print vtbl off`

Do not pretty print C++ virtual function tables.

`show print vtbl`

Show whether C++ virtual function tables are pretty printed, or not.

## Value history

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

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

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

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

If you have a chain of structures where the component `next` points to the next one, you can print the contents of the next one with `p *$.next`.

You can print successive links in the chain by repeating this command, using the Return key.

**NOTE:** The history records values, not expressions. Consider, for instance, if the value of `x` is 4 and you type the following example’s commands.

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

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

`show values`

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

`show values n`

Print ten history values centered on history item number, *n*.

`show values +`

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

Using the Return key to repeat `show values n` has exactly the same effect as ‘`show values +`’.

## Convenience variables

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

Convenience variables are prefixed with ‘\$’. Any name preceded by ‘\$’ can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see “Registers” on page 103). Value history references, in contrast, are *numbers* preceded by ‘\$’. See “Value history” on page 100.

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. For example, set `$foo = *object_ptr` would save in `$foo` the value contained in the object pointed to by `object_ptr`.

Using a convenience variable for the first time creates it, but its value is `void` until you assign a new value. You can alter the value with another assignment at any time. Convenience variables have no fixed types. You can assign a convenience variable any type of value, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

`show convenience`

Print a list of convenience variables used so far, and their values. Abbreviated

`show con.`

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

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

Repeat that command by using the Return key.

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

`$_`

The variable, `$_`, is automatically set by the `x` command to the last address examined (see “Examining memory” on page 91). Other commands which provide a default address for `x` to examine also set `$_` to that address; these commands include `info line` and `info breakpoint`. The type of `$_` is `void *`, except when set by the `x` command, in which case it is a pointer to the type of `$__`.

`$__`

The variable, `$__`, is automatically set by the `x` command to the value found in the

last address examined. Its type is chosen to match the format in which the data was printed.

`$_exitcode`

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



## Registers

You can refer to machine register contents, in expressions, as variables with names starting with ‘\$’. The names of registers are different for each machine; use `info registers` to see the names used on your machine.

`info registers`

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

`info all-registers`

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

`info registers regname...`

Print the *relativized* value of each specified register, *regname*. As discussed in the following, register values are normally relative to the selected stack frame.

*regname* may be any register name valid on the machine you are using, with or without the initial ‘\$’.

GDB has four “standard” register names that are available (in expressions) on most machines—whenever they do not conflict with an architecture’s canonical mnemonics for registers. The register names, `$pc` and `$sp`, are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with `p/x $pc`, or print the instruction to be executed next with `x/i $pc`, or add four to the stack pointer<sup>†</sup> with `set $sp += 4`.

Whenever possible, these four standard register names<sup>‡</sup> are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The `info registers` command shows the canonical names. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`.

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

<sup>†</sup> This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use the Return key; see “Returning from a function” on page 135.

<sup>‡</sup>

Some registers have distinct “raw” and “virtual” data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in “extended” (raw) format, but all C programs expect to work with “double” (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the `info registers` command prints the data in both formats.

Normally, register values are relative to the selected stack frame (see “Selecting a frame” on page 72). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with ‘frame 0’).

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

`set rstack_high_address address`

On AMD 29K family processors, registers are saved in a separate *register stack*. There is no way for GDB to determine the extent of this stack. Normally, GDB just assumes that the stack is large enough to result in GDB referencing memory locations that do not exist. If necessary, you can get around this problem by specifying the ending address of the register stack with the `set rstack_high_address` command. The argument should be an address, which you probably want to precede with ‘0x’ to specify in hexadecimal.

`show rstack_high_address`

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

## Floating point hardware

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

```
info float
```

Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip.

Currently, 'info float' is supported on the ARM and x86 machines.



# Using GDB with different languages

---

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

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

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

- “Switching between source languages” on page 108
- “Displaying the language” on page 110
- “Type and range checking” on page 111
- “Supported languages” on page 114

## Switching between source languages

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

- “List of filename extensions and languages” on page 108
- “Setting the working language” on page 109
- “Having GDB infer the source language” on page 109

In addition to the working language, every source file that GDB knows about has its own working language. For some object file formats, the compiler might indicate which language a particular source file is in. However, most of the time GDB infers the language from the name of the file. The language of a source file controls whether C++ names are demangled—this way `backtrace` can show each frame appropriately for its own language. There is no way to set the language of a source file from within GDB. This is most commonly a problem when you use a program, such as `cfront` or `f2c`, that generates C but is written in another language. In that case, make the program use `#line` directives in its C output; that way GDB will know the correct language of the source code of the original program, and will display that source code, not the generated C code.

### List of filename extensions and languages

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

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

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

CHILL source file.

```
.s
.S
```

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

## Setting 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.

## Having GDB infer the source language

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

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

## Displaying the language

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

`show language`

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

`info frame`

Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See “Information about a frame” on page 74 to identify the other information about the language in the source files.

`info source`

Display the source language of this source file. See “Examining the symbol table” on page 127 to identify the other information about the language in the source files.



## Type and range checking

Some languages are designed to guard against you making seemingly common errors through a series of compile- and run-time checks. These include checking the type of arguments to functions and operators, and making sure mathematical overflows are caught at run time. Checks such as these help to ensure a program's correctness once it has been compiled by eliminating type mismatches, and providing active checks for range errors when your program is running. For more details, see “An overview of type checking” on page 111 and “An overview of range checking” on page 112.

GDB can check for conditions. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation, using the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your program's source language. See “Supported languages” on page 114 for the default settings of supported languages.

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

### An overview of type checking

Some languages, such as Modula-2, are strongly typed, meaning that the arguments to operators and functions have to be of the correct type, otherwise an error occurs. These checks prevent type mismatch errors from ever causing any run-time problems. Consider the following examples.

`1 + 2 ⇒ 3`

Compare with the following example.

`ERROR 1 + 2.3`

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

For the expressions you use in GDB commands, you can tell the GDB type checker to skip checking; to treat any mismatches as errors and abandon the expression; or to only issue warnings when type mismatches occur, but evaluate the expression anyway. When you choose the last of these, GDB evaluates expressions like the second example, but also issues a warning. Even if you turn type checking off, there may be other reasons related to type that prevent GDB from evaluating an expression. For instance, GDB does not know how to add an `int` and a `struct foo`. These particular type errors have nothing to do with the language in use, and usually arise from expressions, such as the one described which make little sense to evaluate anyway. Each language defines to what degree it is strict about type. For instance,

both Modula-2 and C require the arguments to arithmetical operators to be numbers. In C, enumerated types and pointers can be represented as numbers, so that they are valid arguments to mathematical operators. See “Supported languages” on page Supported languages for further details on specific languages.

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

`set check type auto`

Set type checking on or off based on the current working language. See “Supported languages” on page Supported languages, for the default settings for each language.

`set check type on`

`set check type off`

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

`set check type warn`

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

`show type`

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

## An overview of range checking

In some languages (such as Modula-2), it is an error to exceed the bounds of a type; this is enforced with run-time checks. Such range checking is meant to ensure program correctness by making sure computations do not overflow, or indices on an array element access do not exceed the bounds of the array. For expressions you use in GDB commands, you can tell GDB to treat range errors in one of three ways: ignore them, always treat them as errors and abandon the expression, or issue warnings but evaluate the expression anyway. A range error can result from numerical overflow, from exceeding an array index bound, or when you type a constant that is not a member of any type. Some languages, however, do not treat overflows as an error. In many implementations of C, mathematical overflow causes the result to “wrap around” to lower values—for example, if  $m$  is the largest integer value, and  $s$  is the smallest, then the following input is congruent.

$m + 1 \Rightarrow s$

This, too, is specific to individual languages, and in some cases specific to individual

compilers or machines. See “Supported languages” on page Supported languages, for further details on specific languages. GDB provides the following additional commands for controlling the range checker.

`set check range auto`

Set range checking on or off based on the current working language. See “Supported languages” on page Supported languages for the default settings for each language.

`set check range on`

`set check range off`

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

`set check range warn`

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

`show range`

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

## Supported languages

GDB 4 supports C, C++, and Modula-2. Some GDB features may be used in expressions regardless of the language you use: the GDB `@` and `::` operators, and the `{type}addr` construct (see “Expressions” on page 86) can be used with the constructs of any supported language. The following documentation details to what degree each source language is supported by GDB. These sections are not meant to be language tutorials or references, but serve only as a reference guide to what the GDB expression parser accepts, and what input and output formats should look like for different languages. There are many good books written on each of these languages; feel free to use them as a language reference or tutorial in addition to these discussions.

### C and C++

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

The C++ debugging facilities are jointly implemented by the GNU C++ compiler, G++, and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with G++.

For best results when debugging C++ programs, use the `stabs` debugging format. You can select that format explicitly with the G++ command-line options `-gstabs` or `-gstabs+`. See “Options for debugging” on page 103 in *Using GNU CC* in **GNUPro Compiler Tools** for more information.

### C and C++ operators

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

- *Integral types* include `int` with any of its storage-class specifiers; `char`; and `enum`.
- *Floating-point types* include `float` and `double`.
- *Pointer types* include all types defined as `(type*)`.
- *Scalar types* include all of the previous types.

The following operators are supported. They are listed 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  $a \text{ op}= b$ , and translated to  $a = a \text{ op } b$ .  $\text{op}=$  and  $=$  have the same precedence.  $\text{op}$  is any one of the operators  $|$ ,  $\wedge$ ,  $\&$ ,  $\ll$ ,  $\gg$ ,  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ .

? :

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

||

Logical OR. Defined on integral types.

&&

Logical AND. Defined on integral types.

|

Bitwise OR. Defined on integral types.

^

Bitwise exclusive-OR. Defined on integral types.

&

Bitwise AND. Defined on integral types.

==

!=

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

<

>

<=

>=

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

<<

>>

Left shift, and right shift. Defined on integral types.

@

The GDB “artificial array” operator (see “Expressions” on page 86).

+	
-	
	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, '&&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 <code>struct</code> and <code>union</code> data.
[ ]	
	Array indexing. <code>a[i]</code> is defined as <code>*(a+i)</code> . Same precedence as <code>-&gt;</code> .
( )	
	Function parameter list. Same precedence as <code>-&gt;</code> .
::	
	C++ scope resolution operator. Defined on <code>struct</code> , <code>union</code> , and <code>class</code> types. Doubled

colons also represent the GDB scope operator (see “Expressions” on page 86) with the same precedence as the C++ scope resolution operator functionality.

## C and C++ constants

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

- Integer constants are a sequence of digits. Octal constants are specified by a leading ‘0’ (i.e., zero), and hexadecimal constants by a leading ‘0x’ or ‘0X’. Constants may also end with a letter, ‘l’, specifying that the constant should be treated as a `long` value.
- Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: ‘e[+|-]nnn’, where *nnn* is another sequence of digits. The ‘+’ is optional for positive exponents.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (‘’), or a number—the ordinal value of the corresponding character (usually its ASCII value). Within quotes, the single character may be represented by a letter or by escape sequences, which are of the form ‘\nnn’, where *nnn* is the octal representation of the character’s ordinal value; or of the form ‘\x’, where ‘x’ is a predefined special character—for example, ‘\n’ for newline.
- String constants are a sequence of character constants surrounded by double quotes (“ ”).
- Pointer constants are an integral value. You can also write pointers to constants using the C operator, ‘&’.
- Array constants are comma-separated lists surrounded by braces ‘{’ and ‘}’; for example, ‘{1, 2, 3}’ is a three-element array of integers, ‘{{1, 2}, {3, 4}, {5, 6}}’ is a three-by-two array, and ‘{&“hi”, &“there”, &“fred”}’ is a three-element array of pointers.

## C++ expressions

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

**WARNING:** GDB can only debug C++ code if you compile with the GNU C++ compiler, G++. Moreover, C++ debugging depends on the use of additional debugging information in the symbol table, and thus

requires special support. GDB has this support only with the stabs debug format. In particular, if your compiler generates a.out, MIPS ECOFF, RS/6000 XCOFF, or ELF with stabs extensions to the symbol table, these facilities are all available. (With GCC, use the ‘-gstabs’ option to request stabs debugging extensions explicitly.) Where the object code format is standard COFF or DWARF in ELF, on the other hand, most of the C++ support in GDB does not work.

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

```
count = aml->GetOriginal(x, y)
```
- While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer, `this`, following the same rules as C++.
- You can call overloaded functions; GDB resolves the function call to the right definition, with one restriction—you must use arguments of the type required by the function that you want to call. GDB does not perform conversions requiring constructors or user-defined type operators.
- GDB understands variables declared as C++ references; you can use them in expressions just as you do in C++ source—they are automatically dereferenced. In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The address of a reference variable is always shown, unless you have input the ‘set print address off’ command.
- GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in something like an ‘`scope1::scope2::name`’ expression. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see “Program variables” on page 87 for more details).

## C and C++ defaults

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

If you allow GDB to set the language automatically, it recognizes source files whose names end with ‘.c’, ‘.C’, or ‘.cc’, and when GDB enters code compiled from one of



these files, it sets the working language to C or C++. See “Having GDB infer the source language” on page 109 for more details.

## C and C++ type and range checks

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

- The two variables are structured and have the same structure, union, or enumerated tag.
- The two variables have the same type name, or types that have been declared equivalent through `typedef`.

Range checking, if turned on, is done on mathematical operations. Array indices are not checked, since they are often used to index a pointer that is not itself an array.

## GDB and C

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

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See “Expressions” on page 86.

## GDB features for C++

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

`breakpoint menus`

When you want a breakpoint in a function whose name is overloaded, GDB `breakpoint menus` help you specify which function definition you want. See “Breakpoint menus” on page 61.

`rbreak regex`

Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. See “Setting breakpoints” on page 48.

`catch exceptions`

`info catch`

Debug C++ exception handling using these commands. See “Breakpoints and exceptions” on page 53.

`pptype typename`

Print inheritance relationships as well as other information for type `typename`. See “Examining the symbol table” on page 127.

`set print demangle`

`show print demangle`

`set print asm-demangle`

`show print asm-demangle`

Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. See “Print settings” on page 95.

`set print object`

`show print object`

Choose whether to print derived (actual) or declared types of objects. See “Print settings” on page 95.

`set print vtbl`

`show print vtbl`

Control the format for printing virtual function tables. See “Print settings” on page 95.

**Overloaded symbol names**

You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type ‘`symbol(types)`’ rather than just `symbol`. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See “Command completion” on page 25 for details.

## Modula-2

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

## Modula 2 operators

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

- *Integral* types consist of `INTEGER`, `CARDINAL`, and their subranges.
- *Character* types consist of `CHAR` and its subranges.
- *Floating-point* types consist of `REAL`.

- *Pointer* types consist of anything declared as `POINTER TO type`.
- *Scalar* types consist of all of the previous types.
- *Set* types consist of `SET` and `BITSET` types.
- *Boolean* types consist of `BOOLEAN`.

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

'	Function argument or array index separator.
:=	Assignment. The value of <code>var :=value</code> is <code>value</code> .
<	
>	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 “Expressions” on page 86).
+	
-	Addition and subtraction on integral and floating-point types, or union and difference on set types.

<code>*</code>	Multiplication on integral and floating-point types, or set intersection on set types.
<code>/</code>	Division on floating-point types, or symmetric set difference on set types. Same precedence as <code>*</code> .
<code>DIV</code>	
<code>MOD</code>	Integer division and remainder. Defined on integral types. Same precedence as <code>*</code> .
<code>-</code>	Negative. Defined on <code>INTEGER</code> and <code>REAL</code> data.
<code>^</code>	Pointer dereferencing. Defined on pointer types.
<code>NOT</code>	Boolean negation. Defined on boolean types. Same precedence as <code>^</code> .
<code>.</code>	<code>RECORD</code> field selector. Defined on <code>RECORD</code> data. Same precedence as <code>^</code> .
<code>[ ]</code>	Array indexing. Defined on <code>ARRAY</code> data. Same precedence as <code>^</code> .
<code>( )</code>	Procedure argument list. Defined on <code>PROCEDURE</code> objects. Same precedence as <code>^</code> .
<code>::</code>	
<code>.</code>	GDB and Modula-2 scope operators.

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

## Modula-2 built-in functions and procedures

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

<code>a</code>	Represents an <code>ARRAY</code> variable.
<code>c</code>	Represents a <code>CHAR</code> constant or variable.
<code>i</code>	Represents a variable or constant of integral type.
<code>m</code>	Represents an identifier that belongs to a set. Generally used in the same function

with the metavariable,  $s$ . The type of  $s$  should be `SET OF mtype` (where  $mtype$  is the type of  $m$ ).

- $n$  Represents a variable or constant of integral or floating-point type.
- $r$  Represents a variable or constant of floating-point type.
- $t$  “Represents a type.
- $v$  Represents a variable.
- $x$  Represents a variable or constant of one of many types. See the explanation of the function for details.

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

- `ABS( $n$ )`  
Returns the absolute value of  $n$ .
- `CAP( $c$ )`  
If  $c$  is a lower case letter, it returns its upper case equivalent, otherwise it returns its argument
- `CHR( $i$ )`  
Returns the character whose ordinal value is  $i$ .
- `DEC( $v$ )`  
Decrements the value in the variable  $v$ . Returns the new value.
- `DEC( $v$ ,  $i$ )`  
Decrements the value in the variable  $v$  by  $i$ . Returns the new value.
- `EXCL( $m$ ,  $s$ )`  
Removes the element  $m$  from the set  $s$ . Returns the new set.
- `FLOAT( $i$ )`  
Returns the floating point equivalent of the integer  $i$ .
- `HIGH( $a$ )`  
Returns the index of the last member of  $a$ .
- `INC( $v$ )`  
Increments the value in the variable  $v$ . Returns the new value.
- `INC( $v$ ,  $i$ )`  
Increments the value in the variable  $v$  by  $i$ . Returns the new value.
- `INCL( $m$ ,  $s$ )`  
Adds the element  $m$  to the set  $s$  if it is not already there. Returns the new set.

`MAX( t )`

Returns the maximum value of the type *t*.

`MIN( t )`

Returns the minimum value of the type *t*.

`ODD( i )`

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

`ORD( x )`

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

`SIZE( x )`

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

`TRUNC( x )`

Returns the integral part of *x*.

`VAL( t, i )`

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

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

## Modula-2 constants

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

- Integer constants are simply a sequence of digits. When used in an expression, a constant is interpreted to be type-compatible with the rest of the expression. Hexadecimal integers are specified by a trailing ‘H’, and octal integers by a trailing ‘B’.
- Floating point constants appear as a sequence of digits, followed by a decimal point and another sequence of digits. An optional exponent can then be specified, in the form ‘E[+|-]nnn’, where ‘[+|-]nnn’ is the desired exponent. All of the digits of the floating point constant must be valid decimal (base 10) digits.
- Character constants consist of a single character enclosed by a pair of like quotes, either single (‘) or double ("). They may also be expressed by their ordinal value (their ASCII value, usually) followed by a ‘C’.
- String constants consist of a sequence of characters enclosed by a pair of like quotes, either single (‘) or double ("). Escape sequences in the style of C are also allowed. See “C and C++ constants” on page C and C++ constants 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.

## Modula-2 defaults

If type and range checking are set automatically by GDB, they both default to on whenever the working language changes to Modula-2. This happens regardless of whether you, or GDB, selected the working language.

If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with `.mod` sets the working language to Modula-2. See “Setting the working language” on page 109 for further details.

## Deviations from standard Modula-2

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

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

## Modula-2 type and range checks

**WARNING:** In this release, GDB does not yet perform type or range checking. GDB considers two Modula-2 variables type equivalent if the following conditions apply.

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

**NOTE:** This is true of the GNU Modula-2 compiler, but it may not be true of other compilers.) As long as type checking is enabled, any attempt to

combine variables whose types are not equivalent is an error. Range checking is done on all mathematical operations, assignment, array index bounds, and all built-in functions and procedures.

## Modula-2 scope operator (.), the GDB scope operator (::)

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

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

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

Using the Modula-2 operator, ., makes GDB search the current scope for the identifier 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*.

## GDB and Modula-2

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

The @ operator (see “Expressions” on page 86), while available while using any language, is not useful with Modula-2. Its intent is to aid the debugging of *dynamic arrays*, which cannot be created in Modula-2 as they can in C or C++. However, because an address can be specified by an integral constant, the construct ‘{*type*}*adrex*’ is still useful. (see “Expressions” on page 86)

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



# Examining the symbol table

---

The commands described in this section allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB (see "Choosing files" on page 17), or by one of the file-management commands (see "Command files" on page 189).

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see "Program variables" on page 87). 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'` does not work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

`whatis exp`

Print the data type of expression, *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See “Expressions” on page 86.

`whatis`

Print the data type of \$, the last value in the value history.

`ptype typename`

Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form ‘`class class-name`’, ‘`struct struct-tag`’, ‘`union union-tag`’ or ‘`enum enum-tag`’.

`ptype exp`

`ptype`

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

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

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

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with *whatis*, using *ptype* without an argument refers to the type of \$, the last value in the value history.

`info types regexp`

`info types`

Print a brief description of all types whose name matches *regexp* (or all types in your program, if you supply no argument). Each complete *typename* is matched as though it were a complete line; thus, ‘`i type value`’ gives information on all types in your program whose name includes the string *value*, but ‘`i type ^value$`’ gives information only on types whose complete name is *value*.

This command differs from *ptype* in two ways: first, like *whatis*, it does not print a detailed description; second, it lists all source files where a type is defined.

`info source`

Show the name of the current source file—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 the following automatically relinked modules:

```
set symbol-reloading on
```

Replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.

```
set symbol-reloading off
```

Do not replace symbol definitions when re-encountering object files of the same name. This is the default state; if you are not running on a system that permits automatically relinking modules, you should leave *symbol-reloading* off, since otherwise GDB may discard symbols when linking large programs, that may contain several modules (from different directories or libraries) with the same name.

```
show symbol-reloading
```

Show the current on or off setting.

```
maint print symbols filename
```

```
maint print psymbols filename
```

```
maint print msymbols filename
```

Write a dump of debugging symbol data into the file, *filename*. These commands

are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use `'maint print symbols'`, GDB includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols GDB has read.

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 “Commands to specify files” on page 140 for a discussion of how GDB reads symbols (in the description of `symbol-file`).

# Altering execution

---

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program. For example, you can store new values into variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function.

For more information, see the following documentation.

- “Assignment to variables” on page 132
- “Continuing at a different address” on page 133
- “Giving your program a signal” on page 134
- “Returning from a function” on page 135
- “Calling program functions” on page 136
- “Patching programs” on page 137

## Assignment to variables

To alter the value of a variable, evaluate an assignment expression. See “Expressions” on page 86. For example, `print x=4` stores the value 4 into the variable, `x`, and then prints the value of the assignment expression (which is 4). See “Using GDB with different languages” on page 107 for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the expression’s value is not printed and is not put in the value history (see “Value history” on page 100). The expression is evaluated only for its effects.

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

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

The invalid expression, of course, is ‘`=47`’. In order to actually set the program’s variable, `width`, use `(gdb) set var width=47`.

GDB allows more implicit conversions in assignments than C; you can freely store an integer value into a pointer variable or vice versa, and you can convert any structure to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the ‘`{...}`’ construct to generate a value of specified type at a specified address (see “Expressions” on page 86). For example, `{int}0x83040` refers to memory location `0x83040` as an integer (which implies a certain size and representation in memory), and `set {int}0x83040 = 4` stores the value 4 into that memory location.

## Continuing at a different address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the `continue` command. You can instead continue at an address of your own choosing, with the following commands.

`jump linespec`

Resume execution at a specified line, *linespec*. Execution stops again immediately if there is a breakpoint there. See “Printing source lines” on page 78 for a description of the different forms of *linespec*.

The `jump` command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line, *linespec*, is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the `jump` command requests confirmation if the specified line is not in the function currently executing. However, even bizarre results are predictable if you are well acquainted with the machine-language code of your program.

`jump *address`

Resume execution at the instruction at address, *address*.

You can get much the same effect as the `jump` command by storing a new value into the register, `$pc`. The difference is that this does not start your program running; it only changes the address of where it will run when you continue. For example, `set $pc = 0x485` makes the next `continue` command or stepping command execute at address, `0x485`, rather than at the address where your program stopped. See “Continuing and stepping” on page 62.

The most common occasion to use the `jump` command is to back up, perhaps with more breakpoints set, over a portion of a program that has already executed, in order to examine its execution in more detail.

## Giving your program a signal

`signal signal`

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

Alternatively, if *signal* is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinarily see the signal when resumed with the `continue` command; `'signal 0'` causes it to resume without a signal.

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

Invoking the `signal` command is not the same as invoking the `kill` utility from the shell. Sending a signal with `kill` causes GDB to decide what to do with the signal depending on the signal handling tables (see “Signals” on page 65). The `signal` command passes the signal directly to your program.



## Returning from a function

`return`

`return expression`

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

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

This pops the selected stack frame (see “Selecting a frame” on page 72), and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The `return` command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned.

In contrast, the `finish` command (see “Continuing and stepping” on page 62) resumes execution until the selected stack frame returns naturally.

## Calling program functions

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.

`call expr`

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

The user-controlled variable, *call\_scratch\_address*, specifies the location of a scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack does not work in systems that have separate instruction and data spaces.

## Patching programs

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

If you'd like to be able to patch the binary, you can specify that explicitly with the `set write` command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

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

If you specify `'set write on'`, GDB opens executable and core files for both reading and writing; if you specify `'set write off'` (the default), GDB opens them read-only. If you have already loaded a file, you must load it again (using the `exec-file` or `core-file` commands) after changing `set write`, for your new setting to take effect.

```
show write
```

Display whether executable files and core files are opened for writing as well as reading.



## GDB files

---

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, you must also tell GDB the name of the core dump file.

The following documentation discusses more of GDB files.

- “Commands to specify files” on page 140
- “Errors reading symbol files” on page 145

## Commands to specify files

You may want to specify executable and core dump file names. The usual way to do this is at start-up time, using the arguments to GDB's start-up commands (see "Getting in and out of GDB" on page 15).

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 in the following), for more information.

`file`

`file` with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file [filename]`

Specify that the program to be run (but not the symbol table) is found in *filename*. GDB searches the environment variable, `PATH`, if necessary to locate your program. Omitting *filename* means to discard information on the executable file.

`symbol-file [filename]`

Read symbol table information from file, *filename*. `PATH` is searched when necessary. Use the `file` command to get both symbol table and program to run from the same file.

`symbol-file` with no argument clears out GDB information on your program's symbol table. The `symbol-file` command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data

being discarded inside GDB.

`symbol-file` does not repeat if you use `Return` again after executing it once.

When GDB is configured for a particular environment, it understands debugging information in whatever format is the standard generated for that environment; you may use either a GNU compiler, or other compilers that adhere to the local conventions. Best results are usually obtained from GNU compilers; for example, using `gcc` you can generate debugging information for optimized code.

On some kinds of object files, the `symbol-file` command does not normally read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, as they are needed.

The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The `set verbose` command can turn these pauses into messages if desired. See “Optional warnings and messages” on page 184.)

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

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

You can override the GDB two-stage strategy for reading symbol tables by using the ‘`-readnow`’ option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available.

If memory-mapped files are available on your system through the `mmap` system call, you can use another option, ‘`-mapped`’, to cause GDB to write the symbols for your program into a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file (if the program has not changed), rather than spending time reading the symbol table from the executable program.

Using the ‘`-mapped`’ option has the same effect as starting GDB with the ‘`-mapped`’ command-line option.

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

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

host platforms.

`core-file [filename]`

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

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

**NOTE:** The core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the `kill` command (see “Killing the child process” on page 39).

`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.

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

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

When you select remote debugging to a Hitachi SH, H8/300, or H8/500 board (see “GDB and Hitachi microprocessors” on page 171), the `load` command downloads your program to the Hitachi board and also opens it as the current executable target for GDB on your host (like the `file` command).

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

`add-symbol-file filename address`

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

The `add-symbol-file` command reads additional symbol table information from the file, *filename*. You would use this command when *filename* has been dynamically loaded (by some other means) into the program that is running.



*address* should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify *address* as an expression.

The symbol table of the file, *filename*, is added to the symbol table originally read with the `symbol-file` command. You can use the command, `add-symbol-file`, any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `symbol-file` command.

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

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

`add-shared-symbol-file`

The `add-shared-symbol-file` command can be used only under Harris’ CXUX operating system for the Motorola 88k. GDB automatically looks for shared libraries, however if GDB does not find yours, you can run `add-shared-symbol-file`. It takes no arguments.

`section`

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

`info files`

`info target`

`info files` and `info target` are synonymous; both print the current target (see “Specifying a debugging target” on page 147), including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The `help target` command lists all possible targets rather than current ones.

All file-specifying commands allow both absolute and relative file names as arguments. GDB always converts the file name to an absolute file name and remembers it that way.

GDB supports SunOS, SVr4, Irix 5, and IBM RS/6000 shared libraries. GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, remember that GDB does not understand references to a function in a shared library, *unless* you are debugging a core file).

`info share`

## Commands to specify files

---

`info sharedlibrary`

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

`sharedlibrary regex`

`share regex`

Load shared object library symbols for files matching a Unix regular expression.

As with files loaded automatically, it only loads shared libraries required by your program for a core file or after using `run`. If *regex* is omitted, all shared libraries required by your program are loaded.

## Errors reading symbol files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known 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 “Optional warnings and messages” on page 184).

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

`inner block not inside outer block in symbol`

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

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

`block at address out of order`

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

GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. You can often determine what source file is affected by specifying `set verbose on`. See “Optional warnings and messages” on page 184.

`bad block start address patched`

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

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

`bad string table offset in symbol n`

Symbol number *n* contains a pointer into the string table which is larger than the size of the string table. GDB circumvents the problem by considering the symbol to have the name, `f00`, which may cause other problems if many symbols end up with this name.

`unknown symbol type 0xnn`

The symbol information contains new data types that GDB does not yet know

how to read. `0xnn` is the symbol type of the misunderstood information, in hexadecimal.

GDB circumvents the error by ignoring this symbol information. This usually allows you to debug your program, though certain symbols are not accessible. If you encounter such a problem and feel like debugging it, you can debug `gdb` with itself, breakpoint on `complain`, then go up to the function, `read_dbx_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.

## Specifying a debugging target

---

A *target* is the execution environment occupied by your program. Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the `file` or `core` commands. When you need more flexibility—for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection—you can use the `target` command to specify one of the target types configured for GDB

See the following documentation for more discussion of debugging targets.

- “Active targets” on page 148
- “Commands for managing targets” on page 149
- “Choosing target byte order” on page 152
- “Remote debugging” on page 153
- “The GDB remote serial protocol” on page 154

## Active targets

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

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

For example, if you execute `'gdb a.out'`, then the executable file, `a.out`, is the only active target. If you designate a core file as well—presumably from a prior run that crashed and coredumped—then GDB has two active targets and uses them in tandem, looking first in the corefile target, then in the executable file, to satisfy requests for memory addresses. (Typically, these two classes of target are complementary, since core files contain only a program's read-write memory—variables and so on—plus machine status, while executable files contain only the program text and initialized data.)

When you type `run`, your executable file becomes an active process target as well. When a process target is active, all GDB commands requesting memory addresses refer to that target; addresses in an active core file or executable file target are obscured while the process target is active.

Use the `core-file` and `exec-file` commands to select a new core file or executable target (see “Commands to specify files” on page 140). To specify as a target a process that is already running, use the `attach` command (see “Debugging an already-running process” on page 38).

## Commands for managing targets

The following are some commands for targets.

`target type parameters`

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

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

The `target` command does not repeat if you use Return again after executing the command.

`help target`

Displays the names of all targets available. To display targets currently selected, use either `info target` or `info files` (see “Commands to specify files” on page 140).

`help target name`

Describe a particular target, including any parameters (using *name* for the specific target) necessary to select it.

`set gnutarget args`

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

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

**WARNING:** To specify a file format with `set gnutarget`, you must know the actual BFD name.

See “Commands to specify files” on page 140.

`show gnutarget`

Use the `show gnutarget` command to display what file format `gnutarget` is set to read. If you have not set `gnutarget`, GDB will determine the file format for each file automatically and `show gnutarget` displays the following output.

The current BDF target is “auto”.

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

`target exec program`

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

`target core filename`

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

`target remote dev`

Remote serial target in GDB-specific protocol. The argument, *dev*, specifies what serial device to use for the connection (e.g., ‘`/dev/ttya`’). See “Remote debugging” on page 153. `target remote` now supports the `load` command. This is only useful if you have some other way of getting the stub to the target system, and you can put it somewhere in memory where it won’t get clobbered by the download.

`target sim`

CPU simulator. See “Simulated CPU target” on page 175.

`target udi keyword`

Remote AMD29K target, using the AMD UDI protocol. The *keyword* argument specifies which 29K board or simulator to use. See “The UDI protocol for AMD29K” on page 164.

`target amd-eb dev speed prog`

Remote PC-resident AMD EB29K board, attached over serial lines. *dev* is the serial device, as for `target remote`; *speed* allows you to specify the linespeed; and *prog* is the name of the program to be debugged, as it appears to DOS on the PC. See “The EBMON protocol for AMD29K” on page 164.

`target hms dev`

A Hitachi SH, H8/300, or H8/500 board, attached using a serial line to a host. Use special commands, *device* and *speed*, to control the serial line and the communications speed used. See “GDB and Hitachi microprocessors” on page 171.

`target nindy devicename`

An Intel 960 board controlled by a Nindy Monitor. *device-name* is the name of the serial device to use for the connection, e.g. ‘`/dev/ttya`’. See “GDB with a remote i960 (Nindy)” on page 163.

`target st2000 dev speed`

A Tandem ST2000 phone switch, running Tandem’s STD-BUG protocol. *dev* is the name of the device attached to the ST2000 serial line; *speed* is the communication line speed. The arguments are not used if GDB is configured to connect to the ST2000, using TCP or Telnet. See “GDB with a Tandem ST2000” on page 167.



`target vxworks machinename`

A VxWorks system, attached using TCP/IP. The argument, *machinename*, is the target system's machine name or IP address. See “GDB and VxWorks” on page 168.

`target cpu32bug dev`

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

`target op50n dev`

OP50N monitor, running on an OKI HPPA board.

`target w89k dev`

W89K monitor, running on a Winbond HPPA board.

`target est dev`

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

`target rom68k dev`

ROM 68K monitor, running on an IDP board.

`target array dev`

Array Tech LSI33K RAID controller board.

`target sparclite dev`

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

## Choosing target byte order

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

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

---

## Remote debugging

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

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

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

## The GDB remote serial protocol

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

- “What the stub can do for you” on page 155
- “What you must do for the stub” on page 156
- “Putting it all together” on page 158
- “Communication protocol” on page 159
- “Using the gdbserver program” on page 160
- “Using the gdbserve.nlm program” on page 162
- “GDB with a remote i960 (Nindy)” on page 163
- “The UDI protocol for AMD29K” on page 164
- “GDB with a Tandem ST2000” on page 167
- “GDB and VxWorks” on page 168
- “GDB and SPARClet” on page 169
- “Connecting to SPARClet” on page 170
- “SPARClet download” on page 171
- “GDB and Hitachi microprocessors” on page 171
- “GDB and remote MIPS boards” on page 173

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

1. Arrange for all the usual prerequisites for the program to run by itself.  
For example, for a C program, you need the following three prerequisites.
  - A startup routine to set up the C runtime environment; these usually have a name like `‘crt0’`. The startup routine may be supplied by a hardware supplier, so you may have to write your own.
  - You probably need a C subroutine library to support your program’s subroutine calls, notably managing input and output.
  - A way of getting your program to the other machine—for example, a download program. These are often supplied by manufacturers, so you may have to write your own from hardware documentation.
2. Arrange for your program to use a serial port to communicate with the machine

where GDB is running (the *host* machine). In general terms, the scheme follows a standard protocol.

- On the *host*  
GDB already understands how to use this protocol; when everything else is set up, you can simply use the ‘target remote’ command (see “Commands for managing targets” on page 149).

- On the *target*  
You must link with your program using a few special-purpose subroutines that implement the GDB remote serial protocol. The file containing these subroutines is called a debugging stub.

On certain remote targets, you can use an auxiliary program, `gdbserver`, instead of linking a stub into your program. See “Using the `gdbserver` program” on page 160 for details.

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

`sparc-stub.c`

For SPARC architectures.

`m68k-stub.c`

For Motorola 680x0 architectures.

`i386-stub.c`

For Intel 386 and compatible architectures.

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

## What the stub can do for you

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

`set_debug_traps`

This routine arranges for `handle_exception` to run when your program stops.

You must call this subroutine explicitly near the beginning of your program.

`handle_exception`

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

`handle_exception` takes control when your program stops during execution (for example, on a breakpoint), and mediates communications with GDB on the host machine. This is where the communications protocol is implemented; `handle_exception` acts as the GDB representative on the target machine; it begins by sending summary information on the state of your program, then continues to

execute, retrieving and transmitting any information GDB needs, until you execute a GDB command that makes your program resume; at that point, `handle_exception` returns control to your own code on the target machine.

#### `breakpoint`

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

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

## What you must do for the stub

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

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

`int getDebugChar()`

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

`void putDebugChar(int)`

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

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

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

Other routines you need to supply are the following.

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

Write this function to install *exception\_address* in the exception handling tables.

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

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

```
void flush_i_cache()
```

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

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

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

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

This is the standard library function, *memset*, which sets an area of memory to a known value. If you have one of the free versions of *libc.a*, *memset* can be found there; otherwise, you must either obtain it from your hardware manufacturer, or write your own.

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

## Putting it all together

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

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

```
set_debug_traps();  
breakpoint();
```
3. For the 680x0 stub only, you need to provide a variable called `exceptionHook`. Normally you just use `void (*exceptionHook)() = 0;`, but if before calling `set_debug_traps`, you set it to point to a function in your program, that function is called when GDB continues after stopping on a trap (for example, bus error). The function indicated by `exceptionHook` is called with one parameter: an `int` which is the exception number.
4. Compile and link together: your program, the GDB debugging stub for your target architecture, and the supporting subroutines.
5. Make sure you have a serial connection between your target machine and the GDB host, and identify the serial port on the host.
6. Download your program to your target machine (or get it there by whatever means the manufacturer provides), and start it.
7. To start remote debugging, run GDB on the host machine, and specify as an executable file the program that is running in the remote machine. This tells GDB how to find your program’s symbols and the contents of its pure text.

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

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

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

Now you can use all the usual commands to examine and change data and to step and continue the remote program.

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

Whenever GDB is waiting for the remote program, if you use the interrupt character sequence (often, Ctrl-C), GDB attempts to stop the program. This may or may not



succeed, depending in part on the hardware and the serial drivers the remote system uses. If you type the interrupt character once again, GDB displays the following prompt:

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

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

## Communication protocol

The stub files provided with GDB implement the target side of the communication protocol, and the GDB side is implemented in the GDB source file `'remote.c'`. Normally, you can simply allow these subroutines to communicate, and ignore the details. (If you're implementing your own stub file, you can still ignore the details: start with one of the existing stub files. `'sparc-stub.c'` is the best organized, and therefore the easiest to read.)

However, there may be occasions when you need to know something about the protocol—for example, if there is only one serial port to your target machine, you might want your program to do something special if it recognizes a packet meant for GDB.

All GDB commands and responses (other than acknowledgements, which are single characters) are sent as a packet which includes a check-sum. A packet is introduced with the character, `'$'`, and ends with the character, `'#'`, followed by a two-digit checksum: `$packet info#checksum`.

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

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

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

- `g` Requests the values of CPU registers.
- `G` Sets the values of CPU registers.

*maddr, count*

Read *count* bytes at location *addr*.

*Maddr, count:...*

Write *count* bytes at location, *addr*.

*c c addr*

Resume execution at the current address (or at *addr*, if supplied).

*s s addr*

Step the target program for one instruction, from either the current program counter or from *addr*, if supplied.

*k*

Kill the target program.

*?*

Report the most recent signal. To allow you to take advantage of the GDB signal handling commands, one of the functions of the debugging stub is to report CPU traps as the corresponding POSIX signal values.

*T*

Allows the remote stub to send only the registers that GDB needs to make a quick decision about single-stepping or conditional breakpoints. This eliminates the need to fetch the entire register set for each instruction being stepped through.

The GDB remote serial protocol now implements a write-through cache for registers. GDB only re-reads the registers if the target has run.

If you have trouble with the serial connection, you can use the command, `set remotedebug`. This makes GDB report on all packets sent back and forth across the serial line to the remote machine. The packet-debugging information is printed on the GDB standard output stream. `set remotedebug off` turns it off, and `show remotedebug` shows you its current state.

## Using the `gdbserver` program

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

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

`gdbserver` is sometimes useful nevertheless, because it is a much smaller program than GDB itself. It is also easier to port than all of GDB, so you may be able to get started more quickly on a new system by using `gdbserver`. Finally, if you develop code for real-time systems, you may find that the tradeoffs involved in real-time

operation make it more convenient to do as much development work as possible on another system, for example by cross-compiling. You can use `gdbserver` to make a similar choice for debugging.

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

#### *On the target machine*

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

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

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

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

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

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

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

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

#### *On the GDB host machine*

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

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

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

---

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

Its argument is either a device name (usually a serial device like `/dev/ttyb`) or a TCP port descriptor in the form, `host:port`. For example, the input,  
`target remote /dev/ttyb`, communicates with the server via serial line, designated with the pathname, `/dev/ttyb`.

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

## Using the `gdbserve.nlm` program

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

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

### *On the target machine*

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

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

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

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

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

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

## GDB with a remote i960 (Nindy)

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

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

### Startup with Nindy

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

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

Respond to the prompt with whatever suffix (after `/dev/tty`) to identify the serial port that you want to use. You can, if you choose, simply start up with no Nindy connection by responding to the prompt with an empty line. If you do this and later wish to attach to Nindy, use `target` (see “Commands for managing targets” on page 149).

### Nindy reset command

```
reset
```

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

### Options for Nindy

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

```
-r port
```

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

```
-O
```

(An uppercase letter “O”, not a zero.) Specify that GDB should use the “old”

Nindy monitor protocol to connect to the target system. This option is only available when GDB is configured for the Intel 960 target architecture.

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

-brk

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

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

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

## The UDI protocol for AMD29K

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

target udi keyword

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

## The EBMON protocol for AMD29K

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

To use GDB from a Unix system to run programs on the EB29K board, you must first connect a serial cable between the PC (which hosts the EB29K board) and a serial port on the Unix system.

In the following, we assume you’ve hooked the cable between the PC’s ‘`COM1`’ port and ‘`/dev/ttya`’ on the Unix system.

## Communications setup

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

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

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

To give control of the PC to the Unix side of the serial line, at the `c:\>` prompt, type: `CTTY com1`.

(Later, if you wish to return control to the DOS console, you can use the command `CTTY con`—but you must send it over the device that had control, in the example, over the ‘COM1’ serial line). From the Unix host, use a communications program such as `tip` or `cu` to communicate with the PC; for example, `cu -s 9600 -l /dev/ttya`.

The `cu` options shown specify, respectively, the linespeed and the serial port to use. If you use `tip` instead, your command line may look something like: `tip -9600 /dev/ttya`.

Your system may require a different name where we show `/dev/ttya` as the argument to `tip`. The communications parameters, including which port to use, are associated with the `tip` argument in the “remote” descriptions file—normally the system table, `/etc/remotel`.

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

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

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

```
C:\> G:

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

Enter '?' or 'H' for help

PC Coprocessor Type      = EB29K
I/O Base                 = 0x208
Memory Base              = 0xd0000
```

```

Data Memory Size      = 2048KB
Available I-RAM Range  = 0x8000 to 0x1fffff
Available D-RAM Range  = 0x80002000 to 0x801fffff

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

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

# ~.

```

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

## EB29K cross-debugging

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

```

cd /usr/joe/work29k
gdb myfoo

```

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

```

target amd-eb /dev/ttya 9600 MYFOO

```

In this example, we’ve assumed your program is in a file called ‘`myfoo`’.

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

To stop debugging the remote program, use the GDB `detach` command. To return



control of the PC to its console, use `tip` or `cu` once again, after your GDB session has concluded, to attach to `EBMON`. You can then type the command `q` to shut down `EBMON`, returning control to the DOS command-line interpreter. Type `CTTY con` to return command input to the main DOS console, and type `~.` to leave `tip` or `cu`. See Example 1: “PC program, `EBMON` (an `EB29K` control program” on page 165.

## Remote log

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

## GDB with a Tandem ST2000

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

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

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

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

`st2000 command`

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

`connect`

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

## GDB and VxWorks

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

`VxWorks-timeout args`

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

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

## Connecting to VxWorks

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

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

GDB then displays messages like the following declarations.

```
Attaching remote machine across net...
```

```
Connected to tt.
```

GDB then attempts to read the symbol tables of any object modules loaded into the VxWorks target since it was last booted. GDB locates these files by searching the directories listed in the command search path (see "Your program's environment" on page 34); if it fails to find an object file, it displays a message such as:

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

When this happens, add the appropriate directory to the search path with the GDB

command path, and execute the target command again.

## VxWorks download

If you have connected to the VxWorks target and you want to debug an object that has not yet been loaded, you can use the GDB `load` command to download a file from Unix to VxWorks incrementally.

The object file given as an argument to the `load` command is actually opened twice: first by the VxWorks target in order to download the code, then by GDB in order to read the symbol table. This can lead to problems if the current working directories on the two systems differ.

If both systems have had NFS mount the same filesystems, you can avoid these problems by using absolute paths. Otherwise, it is simplest to set the working directory on both systems to the directory in which the object file resides, and then to reference the file by its name, without any path.

For instance, a program `'prog.o'` may reside in `'vxpath/vw/demo/rdb'` in VxWorks and in `'hostpath/vw/demo/rdb'` on the host.

To load this program on VxWorks, type: `-> cd "vxpath/vw/demo/rdb"`

Then, in GDB, type:

```
(vxgdb) cd hostpath/vw/demo/rdb
(vxgdb) load prog.o
```

GDB displays a response similar to the following output.

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

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

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

## Running tasks

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

```
(vxgdb) attach task
```

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

## GDB and SPARClet

GDB enables developers to debug tasks running on SPARClet targets from a Unix

host. GDB uses code that runs on both the Unix host and on the SPARClet target. The program, `gdb`, is installed and executed on the Unix host.

`timeout args`

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

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

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

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

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

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

## Setting file to debug

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

```
(gdb$let) file prog
```

GDB locates the file by searching the directories listed in the command search path. If the file was compiled with debug information (using the option, `-g`), source files will be searched as well. GDB locates the source files by searching the directories listed in the directory search path (see “Your program’s environment” on page 34). If it fails to find a file, it displays a message such as: `prog: No such file or directory`.

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

## Connecting to SPARClet

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

```
target sparclet /dev/ttya
```

GDB displays messages like the following output.

```
Remote target sparclet connected to /dev/ttya
```

```
main () at ../prog.c:3
Connected to tttya.
```

## SPARClet download

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

You'll then see the following output.

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

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

## Running and debugging

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

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

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

```
(gdb:) run
```

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

```
Starting program: prog
Breakpoint 1, main (argc=1, argv=0xffff21c) at prog.c:3
3          char *symarg = 0;
```

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

```
(gdb:) step
4          char *execarg = "hello!";
(gdb:)
```

## GDB and Hitachi microprocessors

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

H8/500.

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

## Connecting to Hitachi boards

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

GDB has another special command to set the communications speed: ‘`speed bps`’. This command also is only used from Unix hosts; on DOS hosts, set the line speed as usual from outside GDB with the DOS `mode` command (for instance, ‘`mode com2:9600,n,8,1,p`’ for a 9600 bps connection).

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

## Using the E7000 in-circuit emulator

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

```
target e7000 port speed
```

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

```
target e7000 hostname
```

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

## Special GDB commands for Hitachi micros

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

```
set machine h8300
```

```
set machine h8300h
```

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

```
set memory mod
```

```
show memory
```

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

## GDB and remote MIPS boards

GDB can use the MIPS remote debugging protocol to talk to a MIPS board attached to a serial line.

This is available when you configure GDB with `--target=mips-idt-ecoff`.

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

```
target mips port
```

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

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

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

```
target mips hostname:portnumber
```

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

GDB also supports the following special commands for MIPS targets.

```
set processor args
```

```
show processor
```

Use the `set processor` command to set the type of MIPS processor when you want to access processor-type-specific registers. For example, the input,

```
set processor r3041,
```

tells GDB to use the CPO registers appropriate for the

3041 chip. Use the `show processor` command to see what MIPS processor GDB is using. Use the `info reg` command to see what registers GDB is using.

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

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

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

```
set remotedebug n
show remotedebug
```

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

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

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

**NOTE:** These commands are available *only* when GDB is configured for `'--target=mips-idt-ecoff'`.

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



## Simulated CPU target

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

`target sim`

Debug programs on a simulated CPU (which CPU depends on the GDB configuration)

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

As well as making available all the usual machine registers (see `info reg`), this debugging target provides three additional items of information as specially named registers:

`cycles`

Counts clock-ticks in the simulator.

`insts`

Counts instructions run in the simulator.

`time`

Execution time in 60ths of a second.

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



# Controlling GDB

---

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, see “Print settings” on page 95; other settings are described in the following documentation.

- “Prompt” on page 178
- “Command editing” on page 179
- “Command history” on page 180
- “Screen size” on page 182
- “Numbers” on page 183
- “Optional warnings and messages” on page 184

## Prompt

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

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

`set prompt newprompt`

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

`show prompt`

Prints a line of the form: 'gdb's prompt is: *your-prompt*'.

## Command editing

GDB reads its input commands via the *readline* interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or vi-style inline editing of commands, *cs*h-like history substitution, and a storage and recall of command history across debugging sessions. You may control the behavior of command line editing in GDB with the command, *set*.

```
set editing
```

```
set editing on
```

Enable command line editing (enabled by default).

```
set editing off
```

Disable command line editing.

```
show editing
```

Show whether command line editing is enabled.

## Command history

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

```
set history filename fname
```

Set the name of the GDB command history file to *fname*. This is the file where GDB reads an initial command history list, and where it writes the command history from this session when it exits. You can access this list through history expansion or through the history command editing characters listed in the following. This file defaults to the value of the environment variable `GDBHISTFILE`, or to `./.gdb_history` if this variable is not set.

```
set history save
```

```
set history save on
```

Record command history in a file, whose name may be specified with the `set history filename` command. By default, this option is disabled.

```
set history save off
```

Stop recording command history in a file.

```
set history size size
```

Set the number of commands which GDB keeps in its history list. This defaults to the value of the environment variable `HISTSIZE`, or to 256 if this variable is not set.

History expansion assigns special meaning to the character, `!`.

Since `!` is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the `set history expansion on` command, you may sometimes need to follow `!` (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities do not attempt substitution on the strings `!=` and `!()`, even when history expansion is enabled.

The commands to control history expansion are the following.

```
set history expansion on
```

```
set history expansion
```

Enable history expansion. History expansion is off by default.

```
set history expansion off
```

Disable history expansion.

The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with GNU Emacs or vi may wish to read it.

```
show history
show history filename
show history save
show history size
show history expansion
```

These commands display the state of the GDB history parameters. `show history` by itself displays all four states.

```
show commands
```

Display the last ten commands in the command history.

```
show commands n
```

Print ten commands centered on command number, *n*.

```
show commands +
```

Print ten commands just after the commands last printed.

## Screen size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Use the Return key when you want to continue the output, or type `q` to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally, GDB knows the size of the screen from the termcap data base together with the value of the `TERM` environment variable and the `stty rows` and `stty cols` settings. If this is not correct, you can override it with the `set height` and `set width` commands:

```
set height lpp
show height
set width cpl
show width
```

These `set` commands specify a screen height of *lpp* lines and a screen width of *cpl* characters. The associated `show` commands display the current settings. If you specify a height of zero lines, GDB does not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer.

Likewise, you can specify '`set width 0`' to prevent GDB from wrapping its output.



## Numbers

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with ‘0’, decimal numbers end with ‘.’, and hexadecimal numbers begin with ‘0x’. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers—when no particular format is specified—is base 10. You can change the default base for both input and output with the `set radix` command.

`set input-radix base`

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

`set output-radix base`

Set the default base for numeric display. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix.

`show input-radix`

Display the current default base for numeric input.

`show output-radix`

Display the current default base for numeric display.

## Optional warnings and messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the `set verbose` command. This makes GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by `set verbose` are those announcing that the symbol table for a source file is being read; see `symbol-file` in “Commands to specify files” on page 140.

```
set verbose on
```

Enables GDB output of certain informational messages.

```
set verbose off
```

Disables GDB output of certain informational messages.

```
show verbose
```

Displays whether `set verbose` is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful (see “Errors reading symbol files” on page 145).

```
set complaints limit
```

Permits GDB to output *limit* complaints about each type of unusual symbols before becoming silent about the problem. Set *limit* to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

```
show complaints
```

Displays how many symbol complaints GDB is permitted to produce.

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

```
(gdb) run
```

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

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

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

```
set confirm off
```

Disables confirmation requests.

```
set confirm on
```

Enables confirmation requests (the default).

```
show confirm
```

Displays state of confirmation requests.

# Canned sequences of commands

---

Aside from breakpoint commands (see “Breakpoint command lists” on page 59), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files. The following documentation provides these discussions for this subject.

- “User-defined commands” on page 186
- “User-defined command hooks” on page 188
- “Command files” on page 189
- “Commands for controlled output” on page 190

## User-defined commands

A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command. This is done with the `define` command. User commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command with `$arg0 ... $arg9`. A trivial example is the following.

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

To execute the command use the following declaration.

```
adder 1 2 3
```

This defines the command, `adder`, printing 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.

## 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 the following debugging input.

```
define hook-stop
handle SIGALRM nopass
end
```

```
define hook-run
handle SIGALRM pass
end
```

```
define hook-continue
handle SIGLARM pass
end
```

You can define a hook for any single-word command in GDB, but not for command aliases; you should define a hook for the basic command name, e.g., `backtrace` rather than `bt`. If an error occurs during the execution of your hook, execution of GDB commands stops and GDB issues a prompt (before the command that you actually used had a chance to run).

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

## Command files

A command file for GDB is a file of lines that are GDB commands.

Comments (lines starting with #) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal. When you start GDB, it automatically executes commands from its *init files*. These are files named `.gdbinit`. GDB reads the init file (if any) in your home directory, then processes command line options and operands, and then reads the init file (if any) in the current working directory. This is so the init file in your home directory can set options (such as `set complaints`) which affect the processing of the command line options and operands. The init files are not executed if you use the `-nx` option; see “Choosing modes” on page 19. 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.

## Commands for controlled output

During the execution of a command file or a user-defined command, normal GDB output is suppressed; the only output that appears is what is explicitly printed by the commands in the definition.

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

`echo text`

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

**NOTE:** No newline is printed unless you specify one.

In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments.

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

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

Consider the following example.

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

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

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

`output expression`

Print the value of *expression* and nothing but that value: no newlines, no ‘\$nn-’. The value is not entered in the value history either. See “Expressions” on page 86 for more information on *expressions*.

`output/ fmt expression`

Print the value of *expression* in format, *fmt*. You can use the same formats as for `print`. See “Output formats” on page 90 for more information.

`printf string, expressions ...`

Print the values of the *expressions* under the control of *string*. The expressions are separated by commas and may be either numbers or pointers. Their values are



printed as specified by *string*, exactly as if your program were to execute the C subroutine, as in the following example.

```
printf (string, expressions...);
```

For example, you can print two values in hex like the following declaration.

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences that you can use in the format string are the simple ones that consist of backslash followed by a letter.



# Using GDB under GNU Emacs

---

GNU Emacs allows you to use, to view and to edit the source files for the program you are debugging with GDB. The following documentation provides information especially for use with the GDB program and the Emacs text editor.

- “Emacs considerations with GDB” on page 194
- “Keystroke sequences for GDB with Emacs” on page 196

## Emacs considerations with GDB

Using GDB under Emacs is just like using GDB normally except for the following considerations.

- All “terminal” input and output goes through the Emacs buffer. This applies both to GDB commands and their output, and to the input and output done by the program you are debugging. This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

Some of the following material uses the convention laid out in *GNU Emacs Manual*<sup>\*</sup>. Meta- signifies using the Meta key (or the diamond ‘◆’ key) on a Unix keyboard and the Alt key on a Windows keyboard in sequence followed by the specified letter. Ctrl- signifies using the Ctrl key in sequence followed by a specified letter. Any other input will be signified by code (as in something typed onscreen like a “gdb” command).

To use the Emacs interface, use the command Meta-x gdb. 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.

All the facilities of Emacs’ Shell mode are available for interacting with your program. In particular, you can send signals the usual way—for example, Ctrl-c, Ctrl-c for an interrupt, Ctrl-c, Ctrl-z for a stop.

- GDB displays source code through Emacs.

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

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

**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

---

<sup>\*</sup> Documentation available from the Free Software Foundation (ISBN 1-882214-03-5).

program resides, or specify an absolute file name when prompted for the **Meta-x** `gdb` argument.

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

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

For example, make Emacs instead call the “*mygdb*” program, using the input, `setq gdb-command-name "mygdb"` (preceded by the keystroke sequence of using the **Esc** key twice consecutively, or by typing in the *\*scratch\** buffer, or in your `.emacs` file).

## Keystroke sequences for GDB with Emacs

In the GDB I/O buffer, you can use the following keystroke sequences of Emacs commands in addition to the standard Shell mode commands.

Ctrl-h, m

Describe the features of Emacs' GDB Mode.

Meta-s

Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

Meta-n

Execute to next source line in this function, skipping all function calls, like the GDB `next` command. Then update the display window to show the current file and location.

Meta-i

Execute one instruction, like the GDB `stepi` command; update display window accordingly.

Meta-x, `gdb-nexti`

Execute to next instruction, using the GDB `nexti` command; update display window accordingly.

Ctrl-c, Ctrl-f

Execute until exit from the selected stack frame, like the GDB `finish` command.

Meta-c

Continue execution of your program, like the GDB `continue` command.

**WARNING:** In Emacs version 19, this command uses the keystroke sequence, Ctrl-c, Ctrl-p.

Meta-u

Go up the number of frames indicated by the numeric argument (see “Numeric Arguments” in *GNU Emacs Manual*), like the GDB `up` command.

**WARNING:** In Emacs version 19, this command uses the keystroke sequence, Ctrl-c, Ctrl-u.

Meta-d

Go down the number of frames indicated by the numeric argument, like the GDB `down` command.

**WARNING:** In Emacs version 19, this command uses the keystroke sequence, Ctrl-c, Ctrl-d.

**Ctrl-x, &**

Read the number where the cursor is positioned, and insert it at the end of the GDB I/O buffer. For example, if you wish to disassemble code around an address that was displayed earlier, type `disassemble`; then move the cursor to the address display, and pick up the argument for `disassemble` by using the keystroke sequence, Ctrl-x, &.

You can customize this further by defining elements of the list `gdb-print-command`; once it is defined, you can format or otherwise process numbers picked up by using the keystroke sequence, Ctrl-x, & before they are inserted. A numeric argument to Ctrl-x, & indicates that you wish special formatting, and also acts as an index to pick an element of the list. If the list element is a string, the number to be inserted is formatted using the Emacs function `format`; otherwise the number is passed as an argument to the corresponding list element.

In any source file, the Emacs command using the keystroke sequence, Ctrl-x, SPACEBAR, and typing `(gdb-break)`, which tells GDB to set a breakpoint on the source line point.

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command, `£`, 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.





# Reporting bugs in GDB

---

Your bug reports play an essential role in making GDB reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. In any case, the principal function of a bug report is to help the entire GNUcommunity by making the next version of GDB work better. Bug reports are your contribution to the maintenance of GDB.

See the following documentation for information for reporting GDB bugs.

- “Have you found a bug?” on page 200
- “How to report bugs” on page 201

Or, if you have the support from Cygnus, use the Cygnus Web Support site:

<http://support.cygnus.com>

## Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the debugger gets a fatal signal, for any input whatever, that is a GDB bug. Reliable debuggers never crash.
- If GDB produces an error message for valid input, that is a bug.
- If GDB does not produce an error message for invalid input, that is a bug. However, you should note that your idea of “invalid input” might be our idea of “an extension” or “support for traditional practice”.
- If you are an experienced user of debugging tools, your suggestions for improvement of GDB are welcome in any case.

## How to report bugs

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

A number of companies and individuals offer support for GNU products. If you obtained GDB from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file `etc/SERVICE` in the GNU Emacs distribution.

In any event, we also recommend that you send bug reports for GDB to one of these addresses:

```
bug-gdb@prep.ai.mit.edu
{ucbvax|mit-eddie|uunet}!prep.ai.mit.edu!bug-gdb
```

**Do not send bug reports to `'info-gdb'` or to `'help-gdb'` or to any newsgroups.** Most users of GDB do not want to receive bug reports. Those who do have arranged to receive `'bug-gdb'`.

The mailing list `'bug-gdb'` has a newsgroup `'gnu.gdb.bug'` which serves as a repeater. The mailing list and the newsgroup carry exactly the same messages. Often people think of posting bug reports to the newsgroup instead of mailing them. This appears to work, but it has one problem which can be crucial: a newsgroup posting often lacks a mail path back to the sender. Thus, if we need to ask for more information, we may be unable to reach you. For this reason, it is better to send bug reports to the mailing list. As a last resort, send bug reports on paper to:

GNU Debugger Bugs  
Free Software Foundation Inc.  
59 Temple Place Suite 330  
Boston, MA 02111-1307 USA

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the debugger into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug if it is new to us. Therefore, always write your bug reports on the assumption that the bug has not been reported previously.

Sometimes people give a few sketchy facts and ask, “Does this ring a bell?” Those bug reports are useless, and we urge everyone *to refuse to respond to them* except to chide the sender to **report bugs properly**.

To enable us to fix the bug, you should include all the following things.

- The version of GDB. GDB announces it if you start with no arguments; you can also print it at any time using `show version`.  
Without this, we will not know whether there is any point in looking for the bug in the current version of GDB.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile GDB—such as “gcc-2.8”.
- What compiler (and its version) was used to compile the program you are debugging—such as “gcc-2.8”.
- 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.

- If you wish to suggest changes to the GDB source, send us context diffs. If you even discuss something in the GDB source, refer to it by context, not by line number.

The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

The following are some things that are not necessary.

- A description of the envelope of the bug.

Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else. Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

Sometimes with a program as complicated as GDB it is very hard to construct an example that will make the program follow a certain path through the code.

If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

Such guesses are usually wrong.

Even we cannot guess right about such things without first using the debugger to find the facts.



# Command line editing

---

The following text describes command line editing interface using Readline library.

- “Readline interaction” on page 207
- “Readline bare essentials” on page 208
- “Readline killing commands” on page 210
- “Readline arguments” on page 211
- “Readline init file” on page 212
- “Variable settings for Readline” on page 213
- “Key bindings for Readline” on page 214
- “Commands for moving around in Readline” on page 215
- “Commands for manipulating history with Readline” on page 216
- “Commands for changing text in Readline” on page 217
- “Killing and yanking” on page 218
- “Specifying numeric arguments” on page 219
- “Letting Readline type for you” on page 220
- “Some miscellaneous Readline commands” on page 221
- “Readline in vi mode” on page 222

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

**IMPORTANT:** The hyphen characters and the comma characters are not a part of the keystroke sequence to type. All uppercase letters require using the shift key, of course, since all commands are specific.

In addition, several keys have their own names. Specifically, Delete, Esc, LFD (linefeed), SPACEBAR, Return, and TAB all stand for themselves when seen in this text, or in an init file (see “Readline init file” on page 212 for more information).



---

## Readline interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply use `Return`. You do not have to be at the end of the line to use `Return`; the entire line is accepted regardless of the location of the cursor within the line.

See the following documentation for more details.

- “Readline bare essentials” on this page
- “Readline movement commands” on page 209
- “Readline killing commands” on page 210
- “Readline arguments” on page 211
- “Letting Readline type for you” on page 220
- “Readline in vi mode” on page 222

## Readline bare essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use **Delete** to back up, and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can use **C-B** to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with **C-F**.

When you add text in the middle of a line, you will notice that characters to the right of the cursor get ‘pushed over’ to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor get ‘pulled back’ to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

**C-B**

Move back one character.

**C-F**

Move forward one character.

**Delete**

Delete the character to the left of the cursor.

**C-D**

Delete the character underneath the cursor.

**Printing characters**

Insert itself into the line at the cursor.

**C-\_**

Undo the last thing that you did. You can undo all the way back to an empty line.

## Readline movement commands

The previous commands are the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to C-B, C-F, C-D, and Delete.

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.

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.

## Readline killing commands

*Killing* text means to delete the text from the line, but to save it away for later use, usually by *yanking* it back into the line. If the description for a command says that it ‘kills’ text, then you can be sure that you can get the text back in a different (or the same) place later. The following is the list of commands for killing text.

### C-K

Kill the text from the current cursor position to the end of the line.

### M-D

Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

### M-Delete

Kill from the cursor to the start of the previous word, or if between words, to the start of the previous word.

### C-W

Kill from the cursor to the previous whitespace.

This is different than M-Delete because the word boundaries differ.

And, here is how to yank the text back into the line.

### C-Y

Yank the most recently killed text back into the buffer at the cursor.

### M-Y

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is C-Y or M-Y.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it in one clean sweep. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

## Readline arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might use M-- C-K.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first ‘digit’ you type is a minus sign (-), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the C-D command an argument of 10, you could use the keystroke sequence, M-1, 0, C-D.

## Readline init file

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

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

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

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

## Variable settings for Readline

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

```
set editing-mode vi
```

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

`editing-mode`

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

`horizontal-scroll-mode`

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

`mark-modified-lines`

This variable when set to `On`, says to display an asterisk, `*`, at the starts of history lines which have been modified. This variable is off by default.

`prefer-visible-bell`

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

## Key bindings for Readline

The syntax for controlling keybindings in the `~/.inputrc` file is simple. First you have to know the name of the command that you want to change. Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the `~/.inputrc` file. The name of the key can be expressed in different ways, depending on which is most comfortable for you. *keyname* and *keyseq* are examples.

*keyname*: *function-name* OR *macro*

*keyname* signifies the name of a key in English. The following text serves as examples.

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

For instance, C-U is bound to the function, `universal-argument`, and C-O is bound to run the macro expressed on the right hand side (that is, to insert the text `'>&output'` into the line).

*"keyseq"*: *function-name* OR *macro*

*keyseq* differs from *keyname* in that strings denoting an entire key sequence can be specified. Simply place the key sequence in double quotes.

GNU Emacs-style key escape sequences can be used, such as in the following examples.

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

For instance, 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 `reread-init-file`, and Esc-[, 1, 1, ~ is bound to insert the text, `Function Key 1`.



## Commands for moving around in Readline

The following documentation contains descriptions of the Readline command name, its default keybinding, and short descriptions of what commands do.

`beginning-of-line` (C-A)

Move to the start of the current line.

`end-of-line` (C-E)

Move to the end of the line.

`forward-char` (C-F)

Move forward a character.

`backward-char` (C-B)

Move back a character.

`forward-word` (M-F)

Move forward to the end of the next word.

`backward-word` (M-B)

Move back to the start of this, or the previous, word.

`clear-screen` (C-L)

Clear the screen leaving the current line at the top of the screen.

## Commands for manipulating history with Readline

The following paragraphs describe the history commands for Readline.

`accept-line` (Newline, Return)

Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history` (C-P)

Move 'up' through the history list.

`next-history` (C-N)

Move 'down' through the history list.

`beginning-of-history` (M-<)

Move to the first line in the history.

`end-of-history` (M->)

Move to the end of the input history, i.e., the line you are entering.

`reverse-search-history` (C-R)

Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history` (C-S)

Search forward starting at the current line and moving 'down' through the the history as necessary.

## Commands for changing text in Readline

The following paragraphs describe commands for changing text in Readline.

`delete-char` (C-D)

Delete the character under the cursor. If the cursor is at the beginning of the line, and there are no characters in the line, and the last character typed was not C-D, then return EOF.

`backward-delete-char` (Rubout)

Delete the character behind the cursor. A numeric argument says to kill the characters instead of deleting them.

`quoted-insert` (C-Q, C-V)

Add the next character that you type to the line verbatim. This is how to insert things like C-Q for example.

`tab-insert` (M-TAB)

Insert a tab character.

`self-insert` (a, b, A, 1, !, ...)

Insert yourself.

`transpose-chars` (C-T)

Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative arguments don't work.

`transpose-words` (M-T)

Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word` (M-U)

Uppercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

`downcase-word` (M-L)

Lowercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

`capitalize-word` (M-C)

Uppercase the first letter in the current (or following) word. With a negative argument, do the previous word, but do not move point.

## Killing and yanking

The following paragraphs describe killing and yanking text.

`kill-line` (C-K)

Kill the text from the current cursor position to the end of the line.

`backward-kill-line` ( )

Kill backward to the beginning of the line. This is normally unbound.

`kill-word` (M-D)

Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`backward-kill-word` (M-Delete)

Kill the word behind the cursor.

`unix-line-discard` (C-U)

Kill the whole line the way C-U used to in Unix line input. The killed text is saved on the kill-ring.

`unix-word-rubout` (C-W)

Kill the word the way C-W used to in Unix line input. The killed text is saved on the kill-ring. This is different than `backward-kill-word` because the word boundaries differ.

`yank` (C-Y)

Yank the top of the kill ring into the buffer at point.

`yank-pop` (M-Y)

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `yank` or `yank-pop`.

## Specifying numeric arguments

The following descriptions are the numeric arguments for Readline.

`digit-argument` (M-0, M-1, ... M--)

Add this digit to the argument already accumulating, or start a new argument. M-- starts a negative argument.

`universal-argument` ( )

Do what C-U does in GNU Emacs. By default, this is not bound.

## Letting Readline type for you

The following documentation details automatic Readline completions.

`complete` (TAB)

Attempt to do completion on the text before point. This is implementation defined. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion.

`possible-completions` (M-?)

List the possible completions of the text before point.

## Some miscellaneous Readline commands

The following documentation details some miscellaneous Readline commands.

`reread-init-file` (C-X, C-R)

Read in the contents of your ‘`~/.inputrc`’ file, and incorporate any bindings found there.

`abort` (C-G)

Stop running the current editing command.

`prefix-meta` (Esc)

Make the next character that you type be metafied. This is for people without a meta key. Typing ESC F is equivalent to typing M-F.

`undo` (C-\_)

Incremental undo, separately remembered for each line.

`revert-line` (M-R)

Undo all changes made to this line. This is like typing the ‘undo’ command enough times to get back to the beginning.

## Readline in vi mode

While the Readline library does not have a full set of vi editing functions, it does contain enough to allow simple editing of the line.

In order to switch interactively between GNU Emacs and vi editing modes, use the command `M-C-J` (`toggle-editing-mode`). When you enter a line in vi mode, you are already placed in ‘insertion’ mode, as if you had typed an ‘i’. Using `Esc` switches you into ‘edit’ mode, where you can edit the text of the line with the standard vi movement keys, move to previous history lines with ‘`k`’, and following lines with ‘`j`’, and so forth.



# Using history interactively

---

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

- “History interaction” on page 224
- “Event designators” on page 224
- “Word designators” on page 224
- “Modifiers” on page 225

## History interaction

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

History expansion takes two parts. In the first part, determine which line from the previous history will be used for substitution. This line is called the *event*. In the second part, select portions of that line for inclusion into the current line. These portions are called *words*. GDB breaks the line into words in the same way that the Bash shell does, so that several English (or Unix) words surrounded by quotes are considered one word.

### Event designators

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

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

### Word designators

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

- 0 (zero)  
The zero'th word. For many applications, this is the command word.
- n  
The *n*'th word.
- ^  
The first argument. that is, word 1.

- `$`  
The last argument.
- `%`  
The word matched by the most recent `?string?` search.
- `x-y`  
A range of words; `-y` abbreviates `0-y`.
- `*`  
All of the words, excepting the zero'th. This is a synonym for `1-$`. It is not an error to use `*` if there is just one word in the event. The empty string is returned in that case.

## Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`.

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



# Formatting documentation

---

The GDB 4 release includes an already-formatted reference card, ready for printing with PostScript or Ghostscript, in the ‘gdb’ subdirectory of the main source directory. If you can use PostScript or Ghostscript with your printer, you can print the reference card immediately with ‘refcard.ps’.

The release also includes the source for the reference card. You can format it, using TEX, by typing: `make refcard.dvi`.

The GDB reference card is designed to print in landscape mode on US “letter” size paper; that is, on a sheet 11 inches wide by 8.5 inches high. You will need to specify this form of printing as an option to your DVI output program. All the documentation for GDB comes as part of the machine-readable distribution. The documentation is written in Texinfo format, which is a documentation system that uses a single source file to produce both online information and a printed manual. You can use one of the Info formatting commands to create the online version of the documentation and TEX (or `texi2roff`) to typeset the printed version. GDB includes an already formatted copy of the online Info version of this manual in the ‘gdb’ subdirectory. The main Info file is ‘gdb-version-number/gdb/gdb.info’, and it refers to subordinate files matching ‘gdb.info\*’ in the same directory. If necessary, you can print out these files, or read them with any editor; but they are easier to read using the `info` subsystem in GNU Emacs or the standalone `info` program, available as part of the GNU Texinfo distribution. If you want to format these Info files yourself, you need one of the Info formatting programs, such as `texinfo-format-buffer` or `makeinfo`.

If you have `makeinfo` installed, and are in the top level GDB source directory, you can make the Info file by typing:

```
cd gdb
make gdb.info
```

If you want to typeset and print copies of this manual, you need TEX, a program to print its DVI output files, and `'texinfo.tex'`, the Texinfo definitions file. TEX is a typesetting program; it does not print files directly, but produces output files called dvi files. To print a typeset document, you need a program to print dvi files. If your system has TEX installed, chances are it has such a program. The precise command to use depends on your system; `lpr -d` is common; another (for PostScript devices) is `dvips`. The DVI print command may require a file name without any extension or a `'dvi'` extension. TEX also requires a macro definitions file called `'texinfo.tex'`. This file tells TEX how to typeset a document written in Texinfo format. On its own, TEX cannot either read or typeset a Texinfo file. `'texinfo.tex'` is distributed with GDB and is located in the `'gdb- version-number/texinfo'` directory. If you have TEX and a dvi printer program installed, you can typeset and print this manual. First switch to the `'gdb'` subdirectory of the main source directory and then type: `make gdb.dvi`.

# Index

---

## Symbols

- !, as not operator 180
- !, as operator for integral types 116
- #, in GDB command files 189
- #, in Modula-2 121
- #define commands 86
- \$ 100, 101
- \$, as variable for registers 103
- \$\_, variable 101
- \$\_, variable 101
- \$\_exitcode, variable 102
- \$cdir 81
- \$cwd 34
- &&, as logical AND operator 115
- &, address operator 116
- &, bitwise AND operator 115
- (), as operators 116
- (), Modula-2 operator for PROCEDURE objects 122
- (gdb) prompt string 178
- (gdb-break) 197
- \*, /, %, as operators 116
- \*, as dereferencing operator 116
- \*, current thread 42
- \*, Modula-2 operator 122
- \*scratch\* buffer in Emacs 195
- ++, --, as increment and decrement operators 116
- +, -, as operators 116
- +, -, Modula-2 operators 121
- ,, Modula-2 operator 121
- ,, sequencing operator 114
- ,=, as operators 115
- , as array indexing operators 116
- , as operator, for integral and floating-point types 116
- , in options 17
- , in options 17
- , Modula-2 array indexing operator 122
- , Modula-2 operator 122
- , Modula-2 operators 121
- , print a newline 190
- ., Modula-2 scope operator 126

- ., ->, as operators 116
- .C 108
- .c 108
- .c++ 108
- .c186 109
- .c286 109
- .cc 108
- .ch 109
- .cp 108
- .cpp 108
- .cxx 108
- .esgdbinit 189
- .gdb\_history 180
- .gdbinit 189
- .mod 108, 125
- .o file 149
- .os68gdbinit 189
- .S 109
- .s 109
- .syms 18
- .vxgdbinit 189
- /, Modula-2 operator 122
- /proc 40
- : , word designator 224
- ::, ,, GDB and Modula-2 scope operators 122
- ::, as C++ scope resolution operator 116
- ::, GDB scope operator 126
- ::, operator 86
- :=, Modula-2 assignment operator 125
- :=, Modula-2 operator 121
- =, , #, Modula-2 operators 121
- =, as operator 115
- =, Modula-2 operators 121
- ==, != , as equality and inequality operator 115
- >, as operators 115
- ?: , ternary operator 115
- @, a binary operator 86
- @, artificial array operator 115
- @, Modula-2 artificial array operator 121
- \_\_raise\_exception 53

- {type}, operator 86
- |, bitwise OR operator 115
- ||, as logical OR operator 115
- ^, bitwise exclusive-OR operator 115
- ^, Modula-2 dereferencing operator 122
- ~, as bitwise complement operator 116
- ~/inputrc 212

## A

- a, Modula-2 variable 122
- a.out, with stabs extensions 118
- ABS, Modula-2 123
- active targets, classes 148
- addr 91
- address ranges 40
- address, locating 90
- add-shared-symbol-file 143
- add-symbol-file 142
- AMD EB29K boards 150
- AMD UDI protocol 150
- AND, &, Modula-2 operators 121
- argument, ignore-count 62
- arithmetical operators 112
- array constants 117
- Array Tech LSI33K RAID controller board 151
- array, artificial 89
- arrays 95
- artificial array 89
- Assembler source file 109
- attach 38
- attach command 148
- automatic display list 93
- awatch 52

## B

- b 19
- backslash-escape sequences 191
- BACKSPACE key 25
- backtrace 71
- batch 19



batch mode 19  
 batch mode for GDB 19  
 --baud option 161  
 BFD 149  
   55  
 Boolean types 121  
 break 67  
 break command 48  
 break...if 49  
 breakpoint 47, 51, 156  
   deleting 54  
   menus 61  
 breakpoint condition 57  
 breakpoint menus 119  
 breakpoint with condition, cond 49  
 breakpoint, enabled or disabled 47  
 breakpoints 41  
   printing a table 50  
 enable 55  
 bt 71  
 bug reports 199

## C

-c 17  
 c 62  
 C and C++ 114  
 C source file 108  
 C++ expressions 117  
 C++ source file 108  
 c, Modula-2 variable 122  
 call stack 69  
 call\_scratch\_address 136  
 calling overloaded functions 118  
 CAP, Modula-2 123  
 catch 53, 119  
 cc -gl on MIPS machines 63  
 C-d 15  
 -cd 19  
 cd 36  
 Character types 120

checksum 159  
 child process 29, 43  
 child process, killing 39  
 CHILL source file 109  
 CHR, Modula-2 123  
 clear command 54  
 -command 17  
 command history file 180  
 command syntax 24  
 commands 59  
 comments (lines starting with #) 189  
 compilation directory 81  
 compile- and run-time checks 111  
 complete 28  
 condition 57  
 confirmation requests 184  
 connect 167  
 continue 62  
 continuing 62  
 convenience variables 101  
 -core 17  
 core dump files 150  
 core-file 142  
 core-file commands 148  
 cpl 182  
 CPU simulator 150  
 CPU time 40  
 CPU32 (M68K) boards 151  
 CPU32BUG monitor 151  
 current thread 41

## D

-d 17  
 d, for deleting breakpoints, watchpoints 54  
 data type 90  
 data types 91  
 debugger, defined 3  
 debugging  
   a29k family 164  
   in a running process 38

- remote 153
- remote serial protocol 154
- specifying a target 147
- stub 155
- symbol file errors 145
- VxWorks targets 168
- with C++ 114
- with GNU C++ compiler, g++ 114
- debugging stub 160
- debugging, when compiling 30
- DEC, Modula-2 123
- declared type 99
- define 186
- delete 54
- delete bnums... 55
- delete display 93
- derived type 99
- detach 38
- detach command 166
- dir 81
- directories, specifying 81
- directory 17
- directory 81
- disable commands 55
- disable display, with expressions 93
- disassemble 82
- disassemblies 120
- display 93
- display/ 93
- DIV, MOD, Modula-2 operator 122
- document 186
- double quotes 117
- down, frame 72
- down-silently, frame 73
- dynamic arrays 126

## E

- e 17
- echo 59, 190
- Emacs 193

- buffer 194
- shell mode 194
- Emacs interface 194
- enable and disable commands 55
- enable commands 55
- enable display 93
- end 59
- end-of-file character, C-d 20
- Enea Data Systems real-time OS 189
- environment 34
- EPROM/ROM code debugging 49
- Ericsson Telecom AB M68000 emulator 189
- ESC, and the '?' key 25
- EST-300 ICE monitor 151
- event designator 224
- examining memory 91
- examining the stack 72
- exception handling 53
- exception handling tables 157
- exceptionHandler 158
- EXCL, Modula-2 123
- exec 17
- exec-file 140
- exec-file commands 148
- executable 149
- executable files 150
- expression, regex 49
- expressions 86

## F

- f 19
- f 72, 74
- fatal signals 65
- fg 62
- file 140, 170
- file-specifying commands 143
- finish 51, 63
- finish command 135
- FLOAT, Modula-2 123
- floating-point registers 103

Floating-point types 120  
 flush\_i\_cache 158  
 -fomit-frame-pointer 70  
 fork 43  
 forward-search 80  
 frame 72, 74  
     Alpha architecture 72  
     MIPS architecture 72  
     SPARC architecture 72  
 frame commands 74  
 frame pointer register 70  
 frames 70  
 Fujitsu SPARClite boards 151  
 -fullname 19

## G

-g option 30  
 GDB 19  
     altering execution 131  
     AMD's 164  
     assignment 132  
     automatic display 93  
     Bash shell 224  
     batch mode 19  
     C, C++ constants 117  
     changing to a different file 140  
     checksum 159  
     command completion with TAB 25  
     command file 189  
     command history facility 180  
     command names, truncated 24  
     commands 159  
     commands for a selected frame 69  
     controlled output 190  
     convenience variables 101  
     converting file names to absolute file names 143  
     core dump 16  
     core dump files 39  
     CPU simulator 175  
     debugging Modula-2 126  
     decimal numbers 183  
     defintion 3  
     delimiters 127  
     EBMON protocol 164  
     environment 31  
     executable files, core files 17  
     exiting 20  
     expression handling 117  
     formatting documentation 227  
     gdbserver 161  
     GNU Emacs 194  
     hexadecimal numbers 183  
     Hitachi SH, H8/300, or H8/500 171  
     Hitachi SH, H8/300, or H8/500 board 142  
     input and output 37  
     Intel 960 142  
     Intel 960 using Nindy 163  
     keystroke sequences of Emacs commands 196  
     language-specific information 107  
     limit complaints 184  
     list or search commands 194  
     MIPS remote debugging protocol 173  
     Motorola 88k 143  
     octal numbers 183  
     operator warning 122  
     operators 86  
     output formats 90  
     parentheses 25  
     patch the binary 137  
     path searches 34  
     print source file and line number 95  
     quotes 26  
     readline interface 179  
     read-only 137  
     remote serial protocol 154  
     reporting bugs 199  
     scope operator, differences with Modula-2 126  
     searches 34  
     sequences for command files 185  
     setting language automatically 109

- shared libraries 143
  - IBM RS/6000 143
  - Irix 5 143
  - SunOS 143
  - SVr4 143
- specify a file 140
- ST2000 environment 167
- standard input and output 31
- starting 17
- starting with run 31
- start-up commands 140
- stop and continue a process 45
- stopping a running program 156
- symbol file errors 145
- terminal modes 37
- type and range checking 111
- user-defined commands 185
- value history 100
- variables 118
- variables, assignment expressions 132
- VxWorks 189
- VxWorks targets 168
- warnings and messages 184
- with C 114
- with C++ 114
- with Modula-2 114
- with TCPconnection 161
- working directory 31, 36
- your program's symbol table 127
- Z8000 family 175
- GDB as a subprocess 19
- GDB commands, command-line options 16
- `gdb -help` 16
- GDB remote serial protocol 161
- GDB, as a filter 19
- GDB, invoking 16
- `gdb`, starting 15
- `gdb/remote.c` 159
- `gdbserve.nlm` 162
- `gdbserver` 160
- `getDebugChar` 158

- `-gg`, for debugging 30
- `-gstabs` 118

## H

- `h` 27
- handle command 65
- handle\_exception 155
- hbreak 49
- help 27
- help target 149
- help user-defined 187
- HIGH, Modula-2 123
- history expansion in `csh` 224
- history numbers 100
- history references 101
- Hitachi SH, h8/300, or h8/500 boards 150
- hooks, user-defined 188

## I

- `i`, Modula-2 variable 122
- `i386-stub.c` 155
- `id` 126
- identifier 126
- IDP board 151
- `if` 186
- IN, Modula-2 operator 121
- INC, Modula-2 123
- INCL, Modula-2 123
- info 28
- info address 127
- info all-registers 103
- info args 74
- info break 50
- info breakpoints 25, 50
- info catch 53, 74, 119
- info display 94
- info `f` 74
- info files 143
- info float, on ARM and x86 machines 105
- info frame 74, 110

info functions 129  
 info line 82  
 info locals 74  
 info proc 40  
 info proc all 40  
 info proc id 40  
 info proc mappings 40  
 info proc status 40  
 info proc times 40  
 info program 46  
 info registers 103  
 info share 143  
 info sharedlibrary 144  
 info signals 65  
 info source 110, 128  
 info sources 129  
 info terminal 37  
 info threads 41  
 info variables 129  
 info watchpoints 50, 52  
 init files 189  
 initial frame 70  
 initialization files, ‘.gdbinit’ 19  
 int getDebugChar() 156  
 Integral types 120  
 Intel 960 boards 150  
 internal breakpoints 51  
 interrupt, C-c 20

## J

jobs -l shell command 38  
 jump command 133

## K

keybindings 212  
 kill 39, 134  
 killing 210

## L

language, displaying source 110  
 linespecs 78  
 list 78  
 list - 79  
 list + 79  
 list commands 24  
 list function 78  
 load 142  
 load using Return 142  
 load, Nindy to Intel 960 142  
 load, VxWorks 142  
 local variables 87  
 longjmp 51  
 longjmp resume 51  
 lpp 182

## M

-m 17  
 m, Modula-2 variable 122  
 m68k-stub.c 155  
 machine instructions 82  
 machine registers 103  
 maint info breakpoints 51  
 maint print msymbols 129  
 maint print psymbols 129  
 maint print symbols 129  
 make 21  
 make refcard.dvi 227  
 -mapped 17  
 MAX, Modula-2 124  
 member function calls 118  
 memory, examining 91  
 memory, examining with x command 91  
 memory-mapped files 17  
 memset 157, 158  
 META key 25  
 MIN, Modula-2 124  
 MIPS  
     function stack 75

- mmap 17
- mode command 172
- Modula-2 107, 120
  - built-in procedures and functions 122
  - C escape sequences 125
  - changes 125
  - constants 124
  - range and type checking 125
  - variables 125
- Modula-2 operators, definitions of types 120
- Modula-2, pointer constants 125
- module 126
- multiple processes 41
- multi-thread programs 41

## N

- n 19
- n, Modula-2 variable 123
- newline 190
- next 63
- nopass 66
- noprint 66
- nostop 65
- NOT, Modula-2 boolean type operator 122
- numbers 183
- nx 19

## O

- O option 30
- object\_ptr 101
- ODD, Modula-2 124
- OKI HPPA board 151
- once bnums... 55
- op=, as operator 115
- OP50N monitor 151
- operators 86
- operators for C and C++ 114
- OR, Modula-2 operator 121
- ORD, Modula-2 124
- outermost frame 70

- output 59
- output format 90
- output, controlled 190
- overloaded symbol 120
- overloading 61

## P

- parent process 43
- pass 66
- path 34
- pointer constants 117
- Pointer types 121
- print 65
- print commands 85
- print settings 95
- print structures 97
- printf 59, 190
- printing with Ghostscript 227
- printing with PostScript 227
- procedure 126
- PROCEDURE objects 122
- process 29
- process group ID 40
- process information 40
- process, stopped 40
- process-id 38
- program counter 72
- prompt 178
- ps utility 38
- ptype 120, 128
- putDebugChar 158
- pwd 36

## Q

- q 19
- quiet 19
- quiet mode 19
- quiet mode for GDB 19
- quit 20

**R**

- r 18
- r 31
- r, Modula-2 variable 123
- range checking 112
- rbreak 49, 119
- readline 207
- readline init file 212
- readnow 18
- register
  - relativized value 103
- register names 103
- register stack 104
- remote debugging 153
- remotedebug variable 174
- reporting a bug 199
- return command 135
- Return key 23
- reverse-search 80
- ROM 68K monitor 151
- RS/6000 xcoff, or elf with stabs extensions 118
- run 31
- run gdb 16
- running process 38
- rwatch 52

**S**

- s 17
- Scalar types 121
- scope 118, 126
- screen size 182
- se 17
- search 80
- searching through a source file 80
- section 143
- session ID 40
- set 28
- set args 33
- set check range auto 113
- set check range off 113

- set check range on 113
- set check range warn 113
- set check type auto 112
- set check type off 112
- set check type on 112
- set check type warn 112
- set complaints 184
- set confirm 184
- set demangle-style 98
- set editing 179
- set gnutarget 149
- set height 182
- set heuristic-fence-post, on MIPS processors 75
- set history 180
- set history filename 180
- set input-radix 183
- set language 109
- set language auto 109
- set language command 108
- set language local 109
- set listsize 78
- set machine h8300 173
- set machine h8300h 173
- set memory 173
- set mipsfpu 174
- set print address 95
- set print address off 95
- set print address on 95
- set print array 96
- set print array off 96
- set print array on 96
- set print asm-demangle 98, 120
- set print asm-demangle on 98
- set print demangle 98, 120
- set print demangle on 98
- set print elements, printing arrays 96
- set print null-stop 97
- set print object 99, 120
- set print object off 99
- set print object on 99

- set print pretty off 97
- set print pretty on 97
- set print sevenbit-strings off 97
- set print sevenbit-strings on 97
- set print symbol-filename off 95
- set print symbol-filename on 95
- set print union 119
- set print union off 97
- set print union on 97
- set print vtbl 99, 120
- set print vtbl off 99
- set print vtbl on 99
- set processor 173
- set prompt command 178
- set remotedebug 174
- set retransmit-timeout 174
- set symbol-reloading off 129
- set symbol-reloading on 129
- set timeout 174
- Set types 121
- set variable command 132
- set verbose 184
- set width 182
- set write off 137
- set write on 137
- set\_debug\_traps 155
- share 144
- shared libraries 143
- sharedlibrary 144
- shell 21
- shell commands 21
- SHELL environment variable 35
- show 28
- show args 33
- show complaints 184
- show convenience 101
- show copying 28
- show directories 81
- show editing 179
- show gnutarget 149
- show height 182
- show history 181
- show language 110
- show listsize 78
- show memory 173
- show mipsfpv 174
- show print address 95
- show print array 96
- show print asm-demangle 98, 120
- show print demangle 98, 120
- show print max-symbolic-offset 96
- show print object 99, 120
- show print pretty 97
- show print sevenbit-strings 97
- show print symbol-filename 96
- show print union 98, 119
- show print vtbl 99, 120
- show processor 173
- show prompt 178
- show range 113
- show remotedebug 174
- show retransmit-timeout 174
- show symbol-reloading 129
- show timeout 174
- show type 112
- show user 187
- show values 100
- show verbose 184
- show version 28
- show warranty 28
- show width 182
- show write 137
- SIGALRM 65
- SIGINT 65
- signal 0 66
- signal command 134
- signal, an asynchronous event 65
- signals
  - fatal 65
- SIGSEGV 65
- SIGTRAP signal 43



- silent 16
- silent 59
- single-stepping 67
- SIZE, Modula-2 124
- source filename 189
- source files 77
- source line 79
- source path 81
- SPARClet
  - connecting to 170
  - loading from a host to debug 171
  - running and debugging 171
  - running on 169
- SPARClite DSU 49
- sparc-stub.c 155
- st2000 167
- stack frame 69
- stack frame, selecting 72
- stack pointer 72
- step 24, 62
- stepi 64
- stepping 62
- stop 65
- string constants 117
- structures 95
- stubs
  - SPARC, M68K, Intel 155
- stubs, defined 154
- style 98
- symbol definitions 143
- symbol file errors 145
- symbol filenames 96
- symbol table 18, 127
- symbol-file 140, 141
- symbols 17
- symbols 95
- systag, thread identifier 41

## T

- t, Modula-2 variable 123

- TAB key 23
- Tandem ST2000 phone switch 150
- Tandem's STD-BUG protocol 150
- target 147, 150
- target amd-eb 150
- target array 151
- target core 150
- target e7000 172
- target est 151
- target exec 150
- target hms 150
- target mips 173
- target nindy 150
- target protocol 149
- target remote 150
- target remote command 158
- target rom68k 151
- target sim 175
- target sparclite 151
- target st2000 150
- target udi 150, 164
- target vxworks 151
- target w89k 151
- target, byte order (MIPS) 152
- targets, active classes 148
- tbreak 49, 56
- TCP port descriptor 162
- Texinfo 227
- thbreak 49
- thread 41, 67
- thread apply 41
- thread command, rejected 41
- thread identifier, systag 42
- threads 41
- timeout 170
- TRUNC, Modula-2 124
- tty 19
- type
  - actual (derived) 99
  - declared 99
- type checking, as for Modula-2 111

types

Modula-2 120

## U

u 63

UDI (Universal Debugger Interface) protocol 164

undisplay 93

union type 119

unset environment 34

until 51, 63

up, frame 72

up-silently, frame 73

user-defined command 186

user-defined hooks 188

using the Return key 23

## V

v, Modula-2 variable 123

VAL, Modula-2 124

value history 90, 100

values, history numbers 100

variable, \$bpnum 48

variable, varname 34

variables 118

variables in expressions 87

virtual function table 99

virtual function tables 120

void \*memset(void \*, int, int) 157

void exceptionHandler 157

void flush\_i\_cache() 157

void putDebugChar(int) 156

VxWorks system 151

VxWorks-timeout 168

## W

W89K monitor 151

warnings and messages 145

watchpoint 47, 51

deleting 54

setting 52

watchpoints

printing a table 50

whatis 128

while 186

Winbond HPPA board 151

Wind River Systems real-time OS 189

## X

-x 17

x command, examine 91

x commands 24

x, Modula-2 variable 123

## Y

yanking 210