Introduction

Cygnus Support Developer's Kit

Cygnus Support Developer's Kit, page 1.
Manuals, page 3.
Using Online Documentation, page 5.
Your Support Contract, page 7.
Reporting Trouble, page 9.
Free Software, page 10.
About Cygnus Support, page 11.
Cygnus Support and the FSF, page 12.

Copyright © 1991, 1992, 1993, 1994, 1995 Cygnus Support

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

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

Cygnus Support Developer's Kit

This Developer's Kit release puts at your disposal, in a single coordinated and tested release, some of the best software development tools available. These tools are available for *native development* (where the same kind of system can run both the tools and the code you develop with them) and for *cross-development* (where one system, the *host*, runs the tools to develop software for another system, the *target*).

Cygnus Support has ported GNU tools to over sixty different platforms. As part of our service, we integrate all our tools to ensure that they all work together. Our developers use a regression testing method that checks not only for problems with a single tool, but across all the tools. We check for problems that a change in one program may cause in another. We don't ship your Developer's Kit release until we know it works.

The Developer's Kit contains these development tools and utilities: *(Not all tools are available for all platforms and operating systems. See section "Overview" in Release Notes, for specific information on your system.)*

Compilers and Development Tools

`gcc` C compiler

`g++` C++ compiler

`gdb` Debugger

`as` Assembler

`gasp` Assembler Preprocessor

`ld` Linker

Libraries

`libg++` C++ class library

`libio` C++ `iostreams` library

`libc` ANSI C runtime library (*only available for cross-development toolkits*)

`libm` C mathematical subroutine library (*only available for cross-development toolkits*)

General Utilities

`byacc` Parser generator

`flex` Fast lexical analyzer generator

`make` Compilation control program

`diff`
`diff3`
`sdiff` Compare text files

`patch` Installs source fixes

`cmp` Compares files byte-by-byte

`send-pr` Sends structured problem reports to Cygnus

`install-sid`
Customizes `send-pr` for your site

`gprof` Performance analyzer (*only available for Sun 4 systems running SunOS 4 or Solaris 2*)

`gcov` Coverage analyzer

Binary Utilities

`c++filt` C++ symbol name deciphering utility

`nm` Lists object file symbol tables

`objdump` Displays object file information

`size` Lists section and total sizes

`ar` Manages object code archives

`ranlib` Generates archive index

`strip` Discards symbols

`objcopy` Copies and translates object files

Text Utilities

Texinfo (requires TEX)
`texindex`
`texi2dvi` Documentation formatting tools

`makeinfo`
`info` Online documentation tools

Manuals

These manuals may be included in a hardcopy format in this Cygnus Support Progressive Release, depending on the nature of your support contract. *Note:* hardcopy versions of the documentation are available as a separate product from Cygnus Support if your contract does not include printed manuals. All documentation is included online with every release; see "Using Online Documentation," page 5.

For convenience we have bound them in two volumes:

Volume I	Volume II
Introduction	*The GNU C++ Iostream Library*
Installation Notes	*The Cygnus C Support Library*
Release Notes	*The Cygnus C Math Library*
Rebuilding From Source	*Using as*
Reporting Problems	*The C Preprocessor*
GNU Online Documentation	*Using ld*
Using GNU CC	*The GNU Binary Utilities*
Debugging with GDB	*Comparing and Merging Files: diff, diff3, sdiff, cmp, and patch*
GNU General Public License	*GNU Make: A Program for Directing Recompilation*

Source for all documentation is also included.

The manuals are designed for easy online browsing (see "Online Documentation," page 5). For online use, the accompanying software distribution includes all the printed manuals, and also the following documents:

FLEX: A Fast Lexical Analyzer Generator
: Generates lexical analyzers suitable for GNU GCC and other compilers.

Using and Porting GNU CC
: Detailed information about what's needed to put `gcc` on different platforms, or to modify `gcc`. Also includes all the information from the printed manual *Using GNU CC*.

BYacc
: A discussion of the Berkeley Yacc parser generator.

User's Guide to the GNU C++ Library
: Details about the general-purpose GNU C++ library, covered under the GNU Library General Public License.

Texinfo: The GNU Documentation Format
: How you can use TeX to print these manuals, and how to write your own manuals in this style.

Cygnus configure
: Details on the configuration program used in Cygnus releases.

GNU Coding Standards
: A complete discussion of the coding standards used by the GNU project.

On the Sun-3 and Sun-4 (SunOS 4.1 or Solaris 2) platforms, the following manual is also provided online:

GNU gprof
: Details on the GNU performance analyzer.

Finally, `man` pages are included for all the programs in the release.

You have the freedom to copy the manuals, like the software they cover; each manual's copyright statement includes the necessary permissions. The manuals themselves are also free software, and the source code for them is also available on the tape.

Conventions

Our manuals use these conventions to help you distinguish commands, filenames, and other program-specific objects from the descriptive text.

`Typewriter-text`
: Indicates text that is a literal example of a piece of a program, such as environmental variable names like `EDITOR`. It will also indicate keyboard characters you should type, or other literal bits of text from a program, such as filenames or examples.

`KEY-NAME`
: Indicates the conventional name for a special key on a keyboard, such as `RET` or `DEL`.

`generic-name`
: Stands for another piece of text. For example, in the command description "To delete the file named *`filename`*, type `rm` *`filename`*." *`filename`* stands for the file you want to delete, no matter what you've named it.

Using Online Documentation

You can browse through the online documentation using either GNU Emacs or the documentation browser program `info` included in the Developer's Kit distribution. Online, the manuals are organized into *nodes*, which correspond to the chapters and sections of a printed book. You can follow them in sequence, if you wish, just like in the printed book—but there are also other choices. The documents have menus that let you go quickly to the node that has the information you need. `info` has "*hot*" references; if one section refers to another, you can tell `info` to take you immediately to that other section—and you can get back again easily to take up your reading where you left off. Naturally, you can also search for particular words or phrases.

The best way to get started with the online documentation system is to run the browser `info`. After this Developer's Kit release is installed on your system, you can get into `info` by just typing its name—no options or arguments are necessary—at your shell's prompt (shown as '`eg%`' here):

```
eg% info
```

(You may need to check that `info` is in your shell path after you install the Developer's Kit release. If you have problems running `info`, please contact your systems administrator.)

To learn how to use `info`, type the command '`h`' for a programmed instruction sequence, or `CTL-h` for a short summary of commands. If at any time you are ready to stop using `info`, type '`q`'.

See section "The Info Program" in *GNU Online Documentation*, for detailed discussion of the `info` program.

Cygnus Support Online Library

All of the manuals in our printed documentation set (see "Manuals," page 3) are also available via the Cygnus Support Information Gallery, our World-Wide Web server, available at

```
http://www.cygnus.com/
```

Contact Cygnus Support for information on connecting via the World-Wide Web.

As with all GNU software, the HTML source for our documents is available (or you can convert them yourself using publicly available utilities) if you wish to put them into an internal Web server for use at your facility. Contact Cygnus Support for details, and please report any problems to the Cygnus documentation department at `doc@cygnus.com`.

Free Software Report

The Free Software Report is a Cygnus publication dedicated to the business of supporting free software.

Volume 1, Number 1: *"Free Software? Yes, Free Software"*

Volume 2, Number 1: *"Free Software And The Law"*

Volume 2, Number 2: *"Free Software Business Models"*

Volume 2, Number 3: *"Free Software: An Agent For Open Systems"*

Volume 3, Number 1: *"Testing, Testing, 1-2-3"*

Inside Cygnus Engineering

Inside Cygnus Engineering is a newsletter describing recent and upcoming activities in the Cygnus Support engineering division. We now have a complete archive of previous issues of *Inside Cygnus Engineering*.

Technical Reports

Cygnus Support technical reports, written by Cygnus engineers.

Security Issues in Embedded Networking

Simple Garbage Collection in G++

The GNU Instruction Scheduler

Runtime Type Support in C and C++

Your Support Contract

GNU development tools provide a powerful, integrated applications development environment. Cygnus Support is one of the primary develpment centers for GNU tools; your support contract links you directly with the developers. With this release, Cygnus Support has provided the latest fully-tested release of the GNU tools, preconfigured for your supported system.

Updates to your progressive release

Every quarter during the period of your support contract, Cygnus Support provides an updated toolchain, complete with the latest enhancements and improvements. The updates also include bug fixes and updated documentation. Most customers receive these updates automatically; if you wish to get automatic updating, please call our support hotline at +1 415 903 1401.

You also receive a monthly newsletter, *Inside Cygnus Engineering,* which keeps you informed about release dates, improvements, new supported platforms, and new products. We often request information about your needs via surveys in this newsletter.

Contacting Cygnus Support

You can reach Cygnus Support by email, phone, or fax. To submit problem reports,

Cygnus Support
toll free: +1 800 CYGNUS-1
main line: +1 415 903 1400
hotline: +1 415 903 1401
email: `support@cygnus.com`

Headquarters	*East Coast*
1937 Landings Drive	48 Grove St., Ste. 105
Mountain View, CA 94043 USA	Somerville, MA 02144 USA
+1 415 903 1400	+1 617 629 3000
+1 415 903 0122 fax	fax +1 617 629 3010

Faxes are answered 8 am–5pm, Monday through Friday.

Future Needs

The availability of source code enables anyone to enhance the GNU tools. While Cygnus is doing the greatest amount of ongoing development, many other users around the world are also contributing enhancements and improvments. Cygnus integration and regression testing ensures that enhancements made elsewhere can work with Cygnus' developments. As your needs evolve, so do the capabilities of the GNU tools and the support services available from Cygnus Support.

Reporting Trouble

We've tried to make the programs in this release of the Cygnus Support Developer's Kit as trouble-free as possible. If you do encounter problems, however, we'd like to diagnose and fix the problem as quickly as possible. You can help us do that by using the script `send-pr` to send us your problem reports (PRs). `send-pr` comes with this release, and is easily configured to send reports back to Cygnus.

`send-pr` invokes an editor on a problem report form (after trying to fill in some fields with reasonable default values). After you exit the editor, `send-pr` sends the filled out form to the Problem Report Management System (PRMS) at Cygnus Support. You can use the environment variable `EDITOR` to specify which editor to use (the default is `vi`). Emacs users will find PRMS especially easy to use.

For more information on `send-pr`, see section "Overview" in *Reporting Problems*.

Filling out a problem report

Problem reports are structured so that a database program can manage them. When you fill out the form, please remember the following guidelines:

- Each PR needs a valid *customer-id* and *category*.
- Describe only one problem per PR.
- For follow-up mail, use the same subject line as the one in the automatic acknowledgment. It shows the category, the PR number and the original synopsis line. This causes your mail to automatically be filed with the original bug report. Your followup comments will be sent to all the people who are working on the bug.
- Please try to make the subject or synopsis line as informative as possible. For misbehaving software, you might use a sentence of the form '`Encrypted rlogin hangs if you send interrupt`' or '`g++: calling wrong overloaded function.`'
- You don't need to delete the comment lines while editing the PR form; this is done by `send-pr`. Put your information before or after the comments.

Consult the section "Examples and guidelines for effective PRs" in *Reporting Problems*, for more discussion on this topic.

Free Software

If you find our Developer's Kit distribution useful, please feel free to give or sell copies of the software and documentation to anyone you like.

In this release, we've assembled the most current editions of these software development tools, tested them, made sure they work well together, and made them easy to install. The installation tape comes with binaries already compiled for your system, and we've made them easy to reconfigure and recompile from source.

These tools are free software; they are part of the GNU project, produced by the Free Software Foundation (FSF). "GNU" is the name of the FSF's evolving operating system (in speech, the 'G' is sounded). Cygnus Support collaborates with the FSF in developing these tools. (For more information on the relationship between Cygnus Support and the FSF, please see "Cygnus Support and the FSF," page 12.)

Cygnus Support exists to help our clients exploit their freedom in using, adapting, or enhancing this software. Cygnus products are *free software*, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt any program it licenses—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must have access to the source code), and the freedom to distribute further copies. Traditional software companies use copyrights to limit your freedoms; the GPL is designed to preserve your freedoms.

Fundamentally, the General Public License is a license which grants you these freedoms, and only imposes restrictions to ensure that no one can take these freedoms away from anyone else.

For full details, see the **LICENSE** section in this manual set, or the file 'COPYING' in the top level of the source code distribution.

About Cygnus Support

Cygnus Support, founded in 1989, provides commercial support for free software. *Why free software?* Free software is fast, powerful, and more portable than its proprietary counterparts. It evolves faster because users who want to make improvements are free to do so.

Cygnus Support has become the leading development organization of the GNU tools, contributing more than 50% of ongoing development. In addition to in-house development, Cygnus leverages the increasing cumulative pool of functionality available as public domain software on the Internet, creating a virtual community of developers. Building upon and contributing to this effort, Cygnus raises the level of functionality available to the entire industry.

The problem with free software has always been that your company's programmers and engineers must spend time maintaining the tools as well as using them, which ties up company resources. Cygnus frees up those resources by supplying products and services, which allows you to use state-of-the-art tools without the problems of maintenance. You can now choose to use free software and get the advantages of powerful and prompt support, combining the best of both.

Our team of experienced engineers include the leading architects for G++, GDB, GAS, and BFD. We use one of the most comprehensive bug-reporting and tracking software in the business, PRMS.

Because all our improvements are also free software, you can distribute them widely within your organization, or to your customers, without extra cost. No unwieldy licenses to manage, and no worries about buying extra copies.

Cygnus makes sure that our customers' problems get solved the right way. No grungy little programs, no klugey fixes that don't generalize, no work-arounds that end up being features instead of temporary situations.

Cygnus Support
toll free: +1 800 CYGNUS-1
main line: +1 415 903 1400
hotline: +1 415 903 1401
email: `support@cygnus.com`

Headquarters	*East Coast*
1937 Landings Drive	48 Grove St., Ste. 105
Mountain View, CA 94043 USA	Somerville, MA 02144 USA
+1 415 903 1400	+1 617 629 3000
+1 415 903 0122 fax	fax +1 617 629 3010

Cygnus Support and the FSF

Cygnus Support and the Free Software Foundation cooperate on major projects, such as the port of the GNU development tools to Solaris 2. We look forward to continuing such cooperation in the future.

Cygnus maintains many programs and libraries *for* the FSF: G++, GDB, GAS, the linker, GPROF, the binary utilities, LIBG++, and the Binary File Descriptor libraries. In addition, Cygnus and the FSF share sources for other programs on a regular basis.

Both Cygnus and the FSF agree that free software is the best way to meet the industry's technical needs. However, Cygnus and the FSF have different missions. Cygnus, as a for-profit company, concentrates on meeting its customers' needs. The FSF concentrates on meeting the internal needs of the GNU project.

Therefore, Cygnus may not accept into our source tree certain FSF changes until they are reworked into a form acceptable for our customers' needs—for example, changes that reduce performance or reliability on the software platforms we support. Conversely, the FSF may not accept changes from Cygnus that improve support for only a limited group of users (Cygnus customers).

Also, Cygnus sometimes supports non-FSF code. Our C subroutine library is an example. The FSF's version carries licensing restrictions which are impractical for customers who write code for embedded systems; as a result, we developed our own subroutine library, LIBC. The Cygnus C subroutine library, although also freely redistributable, does not carry the licensing restrictions that would inconvenience these customers.

Finally, although the FSF makes every effort to provide high-quality releases, its development requirements take priority. The FSF regards software testing as the user community's contribution to the GNU project. Cygnus releases go through a quality assurance cycle which is in large part driven by our knowledge of customer requirements.

Release Notes

Cygnus Support Developer's Kit
Progressive–95q3

Cygnus Support

Edited by Jeffrey Osier (`jeffrey@cygnus.com`).

Copyright © 1995 Cygnus Support

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

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

Table of Contents

Overview **1**
- Checking the Cygnus bug database by mail 2
- Confidential information in problem reports 3

New in this release **4**
- Changes to supported platforms 4
- What's new with GCC 4
- What's new with G++ 5
- What's new with GDB 5
- New options for `ld` 6

Limitations and Warnings **7**
- Programs not available on some platforms 7
- Problem generating PIC under HPUX 7

Issues from previous releases **8**
- Don't use '`-traditional`' with ANSI header files 8
- `-gstabs` gives better C++ COFF debugging 8
- Options for CPU32 and CPU32+ targets 8
- New GCC syntax for function attributes 9
- Debugging remote connections 9
- Accessing the '`m68k-idp`' targets 10
- Leap year calculations 10
- No '`ar q...`' on Unixware 10
- The LANG variable on RS/6000 11
- Requirements for MS-DOS 11
- DEL does not work in MS-DOS Info 11

Notes on rebuilding from source **12**
- Need to upgrade XLC to rebuild for RS/6000 12
- Upgrade `link.h` for Unixware 12
- Use `--with-gnu-as` when configuring MIPS 13
- Rebuilding the tools under Solaris 2 13
- Multiple object code formats 13
- Rebuilding with the GUI debugger 14

Problems fixed in this release **15**
- Binary File Descriptor library 15
- Binary Utilities 15
- Configuration 16

Diff and Patch 16
Flex 16
G++ 16
Assembler 24
Assembler Preprocessor 25
Compiler 25
Debugger 28
Installation 29
Linker 29
C Support Library 30
C++ Support Library 31
Make 31
Motorola Assembler 31
C and Math Support Libraries 32
Problem Reporting 32

Appendix A Graphical User Interface for GDB 33

Getting Started 33
Menus 34
Windows 35

Appendix B Specifying Names for Hosts and Targets 38

Host names 38
Target names 38
`config.guess` 40

Overview

Cygnus Support Developer's Kit Progressive–95q3

The progressive–95q3 release is available for these native and cross-development configurations:

● = available now
○ = contact Cygnus for availability

TARGET ↓ / HOST →	SPARC, Solaris 2.3	SPARC, SunOS 4.1.3	DEC Alpha, OSF/1 v2.0	IBM R/S6000, AIX 3.2.5	IBM R/S6000, AIX 4.1	SGI Iris, Irix 4.05H	SGI Iris, Irix 5.2, 5.3	HP 9000/300, HPUX 9	HP 9000/700, HPUX 9	MS-DOS/Windows/NT	i386, UnixWare 1.1.1
Native	●	●	●	●	○	●	●	●	●		●
Motorola 68k (a.out)	●	●		○				●	●	●	
Motorola 68k IDP	●	●							●	●	
Mot. 68k VxWorks5.1	●	●							●		
Motorola 68k (coff)	●	●							●	●	
PowerPC EABI (elf)	○	●		●	○						
Intel 960 VxWorks5.x	●	●		●					●		
Intel 960 Nindy (coff)		●									
Intel x86 (a.out)		●								●	
AMD 29k VxWorks		●									
AMD 29k UDI (coff)	●	●								●	
SPARC a.out		●								●	
SPARC VxWorks		●									
SPARC v9		○									
SPARClite a.out	●	●								●	
SPARClite coff	●	●								●	
SPARClite VxWorks		○									
MIPS R3000 ecoff/elf	●	●				○				●	
MIPS R4000 elf	●	●				○				●	
Hitachi SH (coff)		●								●	
Hitachi H8/300 (coff)	○	●								●	
Hitachi H8/500 (coff)		○									
HP PRO (elf)	○	○							○	○	
Z8000 coff		○									

Motorola
68000
68020
68030
68040
68060
68EC000
68EC020
68EC030
68EC040
68302
68306
68307
68330
68331
68332
68F333
68F340
68333
68340
68341
68349
68360

Motorola/IBM PowerPC
601
603
604
403GA
403GB

Hitachi
H8/310
H8/322
H8/323
H8/324
H8/325
H8/326
H8/327
H8/328
H8/329
H8/330
H8/350
H8/3332
H8/3002
H8/3003
H8/3040
H8/3041
H8/3042
SH7020
SH7021
SH7032
SH7034
SH7604

MIPS
R4000
R4400
R4600
R4650
R3000A
R3500
R3041
R3051
R3052
R3081
NEC 4200

Fujitsu
930
931
932
933
934

AMD
Am29000
Am29005
Am29050
Am29035
Am29030
Am29040
Am29205
Am29200
Am29245
Am29240
Am29243
Am386SE/DE
Am386SE/LV
Am386EM

SPARC
SPARC V7
SPARC V8
SPARC V9
SuperSPARC

Intel
i386EX
i386CX
i386SX
i486
Pentium
80960CF
80960CA
80960KA
80960KB
80960SA
80960SB
80960HA
80960HD
80960HT
80960JF
80L960JF
80L960JA
80960JA
80960JD

For discussion of the three-part naming scheme used to configure your software for each host/target combination, see Appendix B "Specifying Names for Hosts and Targets," page 38.

These are the current version numbers for the individual programs in the progressive–95q3 release:

Program	*Cygnus Version Numbers*
bfd	2.5-95q3
binutils	2.5-95q3
byacc	+28-95q3
diff	2.6-95q3
expect	5.7.0-95q3
flex	2.4.7-95q3
gas	2.5-95q3
gasp	1.2-95q3
gcc	2.6-95q3
gcov	1.5-95q3
gdb	4.14-95q3
info	2.9-95q3
ld	2.5-95q3
libg++	2.6-95q3
libio	0.67-95q3
libc/libm	1.6-95q3
makeinfo	1.55-95q3
make	3.72-95q3
patch	2.1-95q3
texindex	1.45-95q3
texinfo.tex	2.122-95q3
send-pr	3.2-95q3
tcl	7.3-95q3

Checking the Cygnus bug database by mail

You can interrogate non-confidential bug reports in the Cygnus Problem Report Management System (PRMS) by electronic mail.

Send mail to '`query-pr@cygnus.com`', with query parameters in the '`Subject:`' line of your mail header. (The message body is ignored.)

For example, to inquire about problem reports numbered `4020` and `5004`, send mail including these lines in the header:

```
To: query-pr@cygnus.com
Subject: 4020 5004
```

You can also include many command line options to request information on bugs in a particular state, or a particular category; for example, this header requests a list of all open `G++` bugs that are not confidential:

```
To: query-pr@cygnus.com
Subject: --state=open  --category=g++
```

If you do not include a '`--state=`' specification in your subject line, the mail server uses

```
--state="open|analyzed|feedback|suspended"
```

Careful! Since the default state specification for electronic mail queries does not include `closed`, no news is good news—a closed bug yields a response with no message body.

Also, confidential bug reports are not available via the mail query server. You can request the status of your confidential PRs from your Cygnus technical contact.

Many options are available. To see a synopsis, send a message like this:

```
To: query-pr@cygnus.com
Subject: --help
```

Confidential information in problem reports

There has been some confusion about where to put confidential information in problem reports sent with `send-pr`. If you submit a problem report to Cygnus and you want its detailed contents to remain confidential, set the '`>Confidential:`' field to '`yes`'.

However, the '`Subject:`' line in the mail header and the '`>Synopsis:`' field in the body of the PR are *not* treated as confidential information, as they are used when we compile reports, such as the list of Fixed Problems in this manual (see "Problems fixed in this release," page 15). *Do not* put confidential information in these fields. Any code samples, machine descriptions, problem details, and so on of course remain strictly confidential in any problem report marked as such.

The mail query server for problem reports never reports any information from confidential bug reports.

New in this release

There are improvements in each of the major development tools.

We've also fixed many problems. See "Problems fixed in this release," page 15, for a list of bugfixes.

Changes to supported platforms

HP PA1.1 PRO (`elf`)
: Support for the HP PA1.1 PRO using the `elf` object file format has been expanded. This target is now available for the following hosts:
  - HP9000/700 running HPUX 9 (`hppa1.1-hp-hpux`)
  - Sun SPARCstation running SunOS 4.1.3 (`sparc-sun-sunos4.1.3`)
  - Sun SPARCstation running Solaris 2.x (`sparc-sun-solaris2`)
  - PC running MS-DOS

Intel *x*86 (`elf`)
: The Intel *x*86 chip using the `elf` object file format is now accessible as a target from the Sun SPARCstation running SunOS 4.1.3 (`sparc-sun-sunos4.1.3`).

Intel 960 Nindy monitor (COFF)
: Support for the the Intel 960, COFF object file format, utilizing the Nindy monitor is now available for the Sun SPARCstation running SunOS 4.1.3 (`sparc-sun-sunos4.1.3`).

Hitachi SH3 (COFF)
: Support for the new Hitachi SH3 processor has been added to the SH toolchain.

MIPS r4650
: Support for the r4650 has been added to the MIPS toolchain.

For the complete matrix of supported hosts vs. targets, see "Introduction," page 1.

What's new with GCC

Along with many bug fixes (see "Problems fixed in this release," page 15), the following features have been added to GCC:

- A new option '-fpack-struct' has been added to automatically pack all structure members together without holes.
- Support for the new Hitachi SH3 processor has been added to the SH toolchain.
- Support for the r4650 has been added to the MIPS toolchain.

What's new with G++

A number of major changes appear in this release of G++.

- Support for exception handling has been improved; more targets are now supported, and throws use the RTTI mechanism to match against the catch parameter type. *Warning:* Optimization is not yet supported with '-fhandle-exceptions'.
- Synthesis of compiler-generated constructors, destructors and assignment operators is now deferred until the functions are used.
- Explicit instantiation of template methods is now supported. Also,

  ```
  inline template class foo<int>;
  ```

  can be used to emit only the vtable for a template class.
- The template instantiation code now handles more conversions when passing to a parameter that does not depend on template arguments. This means that code like

  ```
  string s; cout << s;
  ```

  now works properly.
- Invalid jumps in a switch statement past declarations that require initializations are now caught.
- Functions declared 'extern inline' now have the same linkage semantics as inline member functions. On supported targets, where previously these functions (and vtables and template instantiations) would have been defined statically, they are now defined as weak symbols so that only one out-of-line definition is used.

What's new with GDB

Besides many bug fixes this quarter, GDB now features support for the CPU32BUG monitor (found on m68k CPU32 boards, such as the BCC).

Many improvements have also gone into the GDB GUI. The GUI comes in two flavors: the Unix GUI, based on the Tcl/Tk toolkit, and WinGDB, the GUI for use under Microsoft Windows. See Appendix A "Graphical User Interface for GDB," page 33, for details.

New options for `ld`

The GNU linker, `ld`, has two new options:

`-split-by-reloc` *count*
: Attempts to creates extra sections in the output file so that no single output section in the file contains more than *count* relocations. This is useful when generating huge relocatable for downloading into certain real time kernels with the COFF object file format; since COFF cannot represent more than 65535 relocations in a single section. Note that this fails to work with object file formats which do not support arbitrary sections. The linker does not split up individual input sections for redistribution, so if a single input section contains more than *count* relocations one output section contains that many relocations.

`-split-by-file`
: Similar to -split-by-reloc but creates a new output section for each input file.

Limitations and Warnings

Programs not available on some platforms

DEC Alpha running OSF/1 2.0
: The `progressive-95q3` release does *not* include a linker (`ld`) for the Alpha. The native linker is used.

IBM RS/6000 *native*
: The `progressive-95q3` release does *not* include the following supporting programs in the *native* configuration for IBM RS/6000.

`ar`	`objcopy`	`ranlib`
`ld`	`objdump`	`size`
`c++filt`	`nm`	`strip`
`as`		

We ship the GNU compiler configured to use the native linker and assembler, supplied by the machine vendor.

SGI Irix
: The Developer's Kit requires the operating system vendor's C library and include files in a native configuration. The SGI Irix operating system does not contain these files by default, but they are included in a separate developer's package. You cannot use the Cygnus Developer's Kit without this package.

HP9000/700 native
: GNU `ld` is not included for the HP700 in the native configuration.

Problem generating PIC under HPUX

The `gcc` option '`-fPIC`' may cause the compiler to abort when performing function inlining for `hppa1.1-hp-hpux` targets. This problem has been fixed for 95q4.

Issues from previous releases

These issues are not necessarily bugs, but they may help provide advice if you run into problems with our release.

Don't use '`-traditional`' with ANSI header files

Some vendors are starting to ship systems with ANSI C header files. You cannot use '`-traditional`' if you include any header files that rely on ANSI C features.

This is not new behavior, but it can cause confusion. '`-traditional`' causes the compiler to behave as a traditional Kernighan and Ritchie C compiler rather than using the newer ANSI standard.

`-gstabs` gives better C++ COFF debugging

If you use the GCC option '`-gstabs`', GCC embeds extended debugging information in COFF object files. (The extended debug information is based on the *stabs* debugging format, which was originally used only with the `a.out` object file format; see *The stabs debug format*, in your sources as '`src/gdb/doc/stabs.texinfo`', or contact Cygnus for more information.) With this additional debugging information, you can debug C++ programs with GDB, even on systems that use COFF.

You can get better C++ debugging by compiling with '`-gstabs`' for these targets:

```
a29k-amd-udi        m68k-coff
h8300-hms           m88k-coff
z8k-coff            sh-hms
```

Options for CPU32 and CPU32+ targets

The Motorola CPU32 and CPU32+ targets are part of the family of 68000 chips which Cygnus supports. There are a few options to help you compile code for these targets:

- GCC has an option '`-m68332`' to be used specifically when compiling for the Motorola 68332 board. (GCC also has an option '`-m68302`', currently undocumented. The 68302 technically isn't a CPU32 chip.)
- It is also possible to configure GCC for a target of '`m68332-aout`' or '`m68332-coff`' when rebuilding from source, in which case '`-m68332`' is the default.
- GNU AS accepts the following board-specific options:

```
-mcpu32     -m68331     -m68332
-m68333     -m68340     (and -m68302)
```

Contact Cygnus Support for more information on our support for CPU32 and CPU32+ targets.

New GCC syntax for function attributes

To declare that a function does not return, you must now use the '`__attribute__`' keyword (with two leading and trailing underbars) to write something like this:

```
void fatal () __attribute__ ((noreturn));
```

Unfortunately, this new syntax is not compatible with older versions of GCC. Here is an alternative syntax, which works in both the current version and in some older versions:

```
typedef void voidfn ();
...
volatile voidfn fatal;
```

ANSI C does not permit the formerly used syntax, '`volatile void fatal ();`', to have this meaning.

Likewise, to declare that a function has no side effects, so that calls may be deleted or combined, write something like this (which works only with the latest GCC):

```
int computation () __attribute__ ((const));
```

or like this (which works in some older versions):

```
typedef int intfn ();
...
extern const intfn computation;
```

See section "Declaring Attributes of Functions" in *Using GNU CC*, for more discussion of function attributes.

Debugging remote connections

A common hurdle in cross development is to get the communications set up properly between the target board and the development platform.

The GDB '`set remotedebug`' command can help. It was designed to help develop new remote targets; it displays the packets transmitted back and forth between GDB and the target environment. This command can be helpful in diagnosing communications problems, for example allowing you to observe packets not getting through, or noise on the line.

The `set remotedebug` command is now consistent among the MIPS remote target; remote targets using the GDB-specific protocol; UDI (the

AMD debug protocol for the 29k); the 88k BUG monitor; and Hitachi ROM monitors. You can set it to an integer specifying a protocol-debug level (normally `0` or `1`, but `2` means more protocol information for the MIPS target). See section “Communication protocol” in *Debugging with GDB*, for details.

Accessing the ‘`m68k-idp`’ targets

The way GDB accesses programs intended for the IDP board has changed. With this release, as with the previous one, you should compile programs for this board using ‘`-Tidp.ld`’ and the ‘`-nostartfiles`’ option (this option is not necessary for COFF configurations):

```
m68k-coff-gcc ... -Tidp.ld ...
   or
m68k-aout-gcc ... -Tidp.ld -nostartfiles ...
```

This yields a binary file which you can you convert to an S-record using `objcopy`, and then run directly on the IDP board (see section “`objcopy`” in *The GNU Binary Utilities*).

We used to provide pieces of a ROM68K tool chain which were needed to allow GDB to communicate with an IDP board over a serial line. Motorola now supplies a boot monitor called ROM68K with which GDB can communicate. To debug a program compiled in the manner described above, type the following on the GDB command line:

```
target rom68k device 9600
```

where *`device`* corresponds to the device you used to download the binary into the board (for example, ‘`/dev/ttya`’).

Leap year calculations

The calculation in ‘`time.h`’ breaks down whenever a leap year intervenes, though it has been improved in recent months. In particular, leap-seconds are not accounted correctly.

Time functions are in the Cygnus `libc`, supplied for embedded systems only.

No ‘`ar q...`’ on Unixware

The “quick-append” option to GNU `ar`, specified with the keyletter ‘`q`’, does not work on Unixware. Since this option does not conform to the POSIX specification for `ar`, it may be removed altogether in a future release. Use the ‘`r`’ instead. See section “Controlling `ar` on the command line” in *The GNU Binary Utilities*.

The LANG variable on RS/6000

AIX on the RS/6000 provides National Language Support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ('.' vs. ',' for separating decimal fractions). There have been problems reported where the library linked with GCC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the LANG environment variable to either 'C' (the C language locale) or 'En_US' (the American English locale); either value should work.

Requirements for MS-DOS

The Cygnus Developer's Kit is only supported on MS-DOS 6.2 or higher.

We do not recommend using the cross-development kit with less than four (4) megabytes of RAM.

We provide a MS-DOS extender with the cross-development kit for MS-DOS which does swapping to disk when MS-DOS runs out of memory. To avoid excessive swapping you must have at least two (2) megabytes of RAM to run G++ on a PC with MS-DOS. If you've got more than two megabytes, the extra memory can be used as a disk cache to significantly improve performance.

DEL does not work in MS-DOS Info

GNU Info, the online documentation browser, is available in our MS-DOS distribution.

Unfortunately, the DOS version of Info, INFO.EXE, does not recognize the DEL key. This key is normally used for paging backwards within a node in Info.

As a workaround, you can page backwards by keying ESC v.

Notes on rebuilding from source

Details on rebuilding specific platforms and features are shown below. See the manual *Rebuilding From Source* for detailed instructions.

Some general notes:

- You cannot run `configure` in the background; when configuring the program `expect`, `configure` hangs with a terminal message similar to the following:

```
checking mask type of select
checking return type of signal handlers
checking to see if signals need to be re-armed
checking whether cross-compiling... no
checking to see if stty reads from stdin or stdout

[1]  + Suspended (tty output) ../progressive/configu\
re --prefix=`pwd`/stage1 -v |& tee config.out
```

  at which point the shell waits for you to put the process in the foreground again.

Need to upgrade XLC to rebuild for RS/6000

There is a reported problem in rebuilding the Developer's Kit using IBM native tools. (This problem does not crop up if you use GCC to rebuild the tools.)

On the RS/6000, XLC version 1.3.0.0 miscompiles '`jump.c`'. XLC version 1.3.0.1 or later fixes this problem. You can obtain XLC version 1.3.0.2 by requesting PTF 421749 from IBM.

Upgrade `link.h` for Unixware

Note: This problem has been fixed in Unixware 1.1.1 and later.

In order to rebuild the Cygnus Developer's Kit tools from source on Unixware, you must first install a "fixed" '`link.h`' file into '`/usr/include`'. This fix is supposed to be a part of the upcoming *Update 4* for Unixware, and is available on NetWire (the CompuServe support channel for Novell) as well as via anonymous FTP from '`gateway.univel.com`', in the directory '`/pub/developer/`'.

Replace the original '`/usr/include/link.h`' with the new file.

For your reference, Novell's problem ID for this is:

```
dv93-23707     Wrong /usr/include/link.h on Unixware
```

Note: the software *must* be configured with the option '`--with-stabs`' to `configure` when rebuilding the tools for Unixware. This is the default behavior.

For more information, see section “Rebuilding From Source” in *Rebuilding From Source*.

Use `--with-gnu-as` when configuring MIPS

If you rebuild the entire Developer’s Kit from source, the top-level configuration files handle the following configuration detail for you automatically.

But if you rebuild the compiler *alone*, for a MIPS target, we highly recommend that you specify ‘`--with-gnu-as`’ on the command line for `configure`. This avoids an incompatibility between the GNU assembler and the MIPS assembler. The MIPS assembler does not support debugging directives, and GCC uses a special program, `mips-tfile`, to generate them. GNU AS parses the debugging directives directly, and does not require `mips-tfile`.

You should also specify ‘`--with-stabs`’ on the command line to `configure`. This provides better debugging symbols, in particular for C++.

If you plan to use GNU LD, be sure to specify ‘`--with-gnu-ld`’ when you rebuild on any platform for which the linker is available.

Rebuilding the tools under Solaris 2

If you wish to rebuild the tools from source on your SPARC system running Solaris 2, you can use either the original Solaris 2 native-development binaries from the Cygnus Support Developer’s Kit or the unbundled compiler sold separately by Sun.

Beware! You might notice that there is a program called `/usr/ucb/cc`, and be tempted to use it. Don’t; this program is incompatible with the *real* compiler, which is in ‘`/opt/SUNWspro/bin/cc`’.

For more information on rebuilding the Developer’s Kit, see section “Rebuilding From Source” in *Rebuilding From Source*.

Multiple object code formats

As in previous releases, you can reconfigure the Developer’s Kit tools to support more than one object format (see section “Rebuilding From Source” in *Rebuilding From Source*). However, we’ve changed the `configure` command-line option slightly.

To add support for more object file formats (besides the format appropriate for the configured target), list the additional targets as arguments

to the `configure` option '`--enable-targets`', separated by commas. For example,

```
./configure --enable-targets=m68k-coff,i386-elf,decstation
```

To find out what targets are available, look in the file '`bfd/config.bfd`' in the source distribution.

To configure the Developer's Kit tools to support all available object formats, use '`--enable-targets=all`' rather than listing individual targets.

Rebuilding with the GUI debugger

If you are rebuilding from source and wish to configure GDB to use the new Tk-based GUI (see Appendix A "Graphical User Interface for GDB," page 33), you must use the option `--enable-gdbtk` on the command line to `configure`.

Note: Using '`--enable-gdbtk`' when rebuilding under Solaris or HPUX is known to fail. Contact Cygnus for an available patch if this is a problem for you. (Reference PR #6743)

For more information on rebuilding from source, see the manual *Rebuilding From Source*.

Problems fixed in this release

Here is a list of the problems we have fixed since the last Progressive release. We hope that you find it useful. (You can contact us at +1 415 903 1401 to inquire about the status of any problems.)

This information, as well as a list of all problems that have been reported to us that are still outstanding, is available in ASCII form from Cygnus Support. Contact us at `support@cygnus.com` or at the phone number above and ask for the ASCII list of known and/or fixed bugs for 95q3.

The following summaries of fixed bugs in the `progressive-95q3` release are organized by the reporting category—that is, by the software component, such as `gdb` or `g++`.

Each bug summary begins with the Cygnus Support Problem Report number. We consider a problem report *closed* only when the customer who reported the bug agrees the problem is solved.

Binary File Descriptor library

2419 GCC generates wrong magic number for mc68000

4083 bfd/archive.c needs tweak to work with hp-pa for s300 cross

4854 no way to tell relocatable from absolute object file

Binary Utilities

3611 Suggested option for nm

4114 Disassembler does not handle BASE+DISPLACEMENT address (I3008)

4449 sparc-x-386lynx strip doesn't remove enough

4779 objdump number display base not consistent

5057 m68k-aout-objdump -d generates a 'tpcc #7' instead of 'trap #7'

5059 objdump -d could be more useful

5859 ar breaks purify

6024 ld fails on pic objects not in shared library

6255 ar tv nulls out oversized filenames instead of truncating

6345 all binutils should have an option to display BFD backends

6391 ar tv dumps core

6540 pb with library (creation)

6551 -static flag generated an internal compiler error [ld error]

6660 C++ filter doesn't seem to demangle virtual tables

6736 GAS-SH crashes on undefined symbols

6985 objdump does not display

7023 29k COFF to Hex tools

Configuration

2086 error message for missing Makefile.in

2888 configure –srcdir=‘pwd‘ eats files

6389 configure fails to pass down –config-cache=FILE arg

6576 config.guess doesn't match progressive precompiles.

6593 Why don't you provide a binary for 'protoize'?

Diff and Patch

3235 Diff doesn't compile with cc due to stray `"const"`

Flex

5971 flex requires libg++

G++

515 name of virtual is converted to ptr to a method, no vtbl indirection

1018 segmentation violation caused by ambiguousness of conversions

1070 PT parsing bugs

1205 question on building program that uses templates

1248 pure virtual destructor causes abend

1375 GNU C++, failed assertion in cp-lex.c:1109

1382	GNU C++ crashes when compiling a const member function
1416	Can't invoke [cg]++68k,sparc
1419	question about browsing classes
1460	GNU C++ doesn't do the right thing with -nostdinc
1462	new with placement syntax
1497	cc1plus gets fatal signal
1514	g++ doesn't provide struct debugging information
1533	g++ instantiates some method templates it shouldn't
1553	fatal signal from c++ compiler
1583	question on pure virtual destructor
1640	"-a" for test coverage doesn't work
1657	g++ core dumps under error conditions
1772	expressions on enums of two elements become of type int
2124	Have a question about how G++ handles new of arrays
2153	compiler get fatal signal 11 (segmentation violation)
2279	gdb 4.8 can't print struct/class with no member functions
2374	g++ stabs generation prevents debugging
2418	g++ fails to accept multiple extern declarations
2444	question on include directories automatically searched by g++/gcc
2513	g++ code gets segmentation fault
2519	g++ produces incorrect results with -O or -O2
2572	Compiler errors, using inheritance
2608	confused vitrual pointer tables (OR confused programmer: me)
2747	bad code is generated for delete [] of types without destructors
2761	inconsistent return types error for template class constructor
2984	friend Template operators dont work unless they are inline
3006	cannot convert int to const char &
3015	g++ calls wrong virtual function
	3045 g++ does not get “error: jump past initializer (did you forget a ' '?)” but Sun CC does

3150 Internal error in H8/g++ 3190 error “signment of read-only variable ‘d2’ ” should say “assignment of read-only variable IN variable ‘d2’ ”

3242 another template “syntax error at null character” reparsing error

3244 no error if template member function occurs after template instance use

3265 template bug: as error generated

3295 g++ calls the wrong virtual functions

3572 Bogus warning from g++

3587 Bizarre errors for nested classes in a template

3613 CC allows global scope operator on class names but g++ does not

3634 forward declaration of template class after actual declaration

3639 Compilation error associated with -Woverloaded-virtual

3681 Accurate line number desired for “MapCH.h:319: syntax error before ‘*’ ”

3852 -Wenum-clash not included in -Wall

3911 can we do better than “syntax error before ‘*’ ” here?

4001 operator delete cannot be overloaded

4078 compiler emits warnings unnecessarily, aborts with math-68881.h

4121 Memory fault when using cerr in class constructor.

4180 Incorrect warning statement.

4258 problem with an array class

4278 further thoughts on bug reported yesterday re: overloaded operators

4341 Code for some member functions is not generated

4464 get seemingly erroneous error message from g++

4465 string.h declarations are inconsistent with builtins

4588 inconsistent behaviour of g++ using ’++’-operators (prefix/postfix)

4627 ‘void foo() static const char *const v[] = 0 ; ’ gets internal error

4633 template argument with explicit global scope: g++ gives error

4634 Pointers to member function, overloading: g++ gives wrong error

4646 friend function in a template class: g++ ld error

4647 template class, forward declaratoin of a template function

4650 template class

4651 template class

4653 template class, pointer to member function

4655 template class, template arg

4656 Function templates

4659 template function

4660 template class, constructors

4661 template class, friend class

4662 template class, default argument in constructor/destructor

4664 class, C++ function, extern C functon with the same name

4665 Static class object initializing

4666 Scope reference inside the same class

4667 Virtual bases constructed in wrong order

4668 Internal compiler error.

4679 g++ acts strangely toward forward declaration for template class

4684 extern "C" friend function

4685 extern "C" friend function, wrong duplicate declaration

4686 Template member functions not being generated in g++

4688 virtuals, templates, and C, oh my!

4692 g++ allows foo->member and foo.member to be applied when foo is a pointer to an object

4695 should allow passing reference to undefined object

4696 const objects w/o constructors shouldn't always go in text segment

4707 parser problem

4738 GDB's ptype won't pierce structure

4745 g++ complains of inconsistent return type for template constructor

4747 static class members not generated in g++

4756 Cygnus Release Doesn’t Work With Purify/Quantify

4790 linker error when using -O

4819 global constructor from library not getting run

4866 template instantiation facilities are inadequate

4907 erroneous ’too many arguments for method’ message

4917 1 of n static members in a class is undefined.

4969 AIX ld too "smart" ?

4972 virtual function declared but not defined gets error

4977 inheritance, default copy constructor

5028 global constructors are called twice each

5126 C++ constructor produces bad code

5134 g++ generates incorrectly aligned code with MI and virtual bases

5148 internal compiler error, pointers to member functions

5158 Possibly bogus aritmetic overflow warning

5218 #pragma implementation makes *all* funcs global

5228 more __attribute__ ((packed)) bugs

5266 “#pragma implementation "NOTHING"” is silently ignored if it is used too far into the file (no warning or error)

5306 dtor called for never constructed object

5307 eof not set

5382 g++ gets very confused by matching declarations and definitions with default args and gives an error

5388 g++’ concept of the known size of an initializer differs here – and the error messages are real cryptic

5407 cc1plus can’t deal with this file

5459 FSF 2.6.0 has lost the ability to reference lexically visible static class members in a template

5462 Cannot declare a destructor in a user-provided class template instantiation

5463 gratuitous ‘value computed is not used’ may hide further badness?

5468 gratuitous? “redeclaration of derivation chain of type ‘class MapLS<String,Ref<X3> >’”

5470 Why are the synthetic member functions with internal linkage, copied in each file?

5472 large char array causes slow compilation

5503 g++ gets confused about how many arguments an imported nested class needs

5526 cxx requires 800K to compile this while g++ requires 20MB; cxx generates 2K of code; g++ generates 11K

5541 something causes a core dump

5588 g++ gets an internal compiler error with calls to the destructor for template types, when the type is a pointer to member function.

5611 virtual function in local class: internal compiler error

5615 Destructor called twice here when a user-defined conversion is silently invoked

5622 declaration causes syntax error

5624 Linker complains about missing inlined template methods

5643 error on “initialization to ‘void (X::*)()’ from ‘void (X::*)()’ ”

5659 g++ gives wrong error on operator overloading

5666 g++ deletes temporaries before they should be deleted

5668 g++ gives wrong error (function template)

5671 Link problem

5684 g++ doesn’t generate valid temporary object

5692 the message “ argument list for ‘c’ does not match any in class ‘A::c’ ” is misleading in this instance

5745 g++ gives wrong error dealing with local class

5763 overload operator resolution: g++ gives wrong error

5959 internal error on scalar init for array.

6032 g++ fails to access protected member of parent’s parent class when parent is a virtual base

6037 g++ calls constructor for an object even if new fails

6039	GCC_EXEC_PREFIX conflicts with Purify Software tools
6047	g++ can't mangle names correctly where part of the name is an anonymous type
6050	How can one explicitly call static initializers from dls under sunos?
6057	new compile error 1st compile after upgrade
6082	gcc uses C include path for C++ includes
6093	operator delete access control not enforced
6131	unable to compile constructors of template that inherits from virtual base class
6151	using -fno-implicit-templates causes pc-relative reference to external symbol on 29k target
6172	syntax errors when new operator is overloaded
6193	overloading of new with a class encounters syntax problems
6205	pragma implementation used to be default if base matched header
6261	Need help with undefined symbols.
6330	new syntax errors on previously compiled code with new compiler
6553	Is 'true' now a reserved C++ keyword?
6568	Overload resolution of user defined operators broken between 2.6.3 and rotd Feb 23
6569	MI, virtual bases and rtti
6582	g++ treats enum as UNsigned
6609	Not all templates are exported with -fexternal-templates
6610	Virtual base classes fail to allocate properly
6625	Missing declarations
6629	abs(double) redefinition whines
6648	Missing operator= for ostream
6658	g++ - type checking failure - ostream& v/s ostream*
6661	Internal compiler error for static initializers
6662	gcc: Internal compiler error
6663	I960 branch prediction bit

6683 Internal compiler error + Compiler hang

6704 internal compiler error

6705 compiler hangs

6710 compiler hangs

6721 fixproto fails to parse some header files

6802 Templates ain't working

6816 g++: array-ptr-conversion

6835 c++ reports "internal compiler error" in function almostEq

6836 when I compile this example, I get a compiler error in std. includes.

6847 Cannot successfully apply previously received patch.

6885 Incorrect type handling

6908 g++ generates an invalid access control error.

6909 Overriding virtual function returning different type loses

6917 virtual table thingy

6943 #pragma interface causes undefined references to virtual tables

6979 Getting "warning: recoverable compiler error, fixups for virtual function"

6980 we have a problem to report

7007 When is the explicit keyword functionality due out?

7013 g++ 95q2 can't generate static (non-public) vtables

7019 g++ : internal compiler error

7028 g++ : Internal compiler error

7037 Re: Question regarding g++/6981

7068 g++ 29k incorrectly overwrites a register used to cache a function address

7077 g++ applies conversions differently between infix and prefix form of the same operator

7095 request to ftp latest ANSI C++ paper if you have it

Assembler

1358 as68k accepts "-m68881"

1359 gas -mc68000 causes a bus error on DECstation with floating point insns

1360 gas for sparc doesn't recognize "trivial save"

1368 gas: doesn't know about fpcr,fpsr,fpiar

1376 gas-sparc: some synthetic instructions don't support constants

1480 68020 assembly instruction not handled by gas

1664 lables resembling registers confuse as68k

1673 The size of the bsr instruction varies with the displacement

1674 m68k-coff-as blows up but m68k-aout-as doesn't

1741 Assembler confused about symbols starting with "sp."

2071 gas assembly listing format error

2434 gas generates incorrect addresses for pc-relative jsr

2663 gas produces wrong code for label(pc,d7.w)

3211 Assistance With gas

3292 need software workaround for 68331 processor hardware bug

3554 using the MARK macro causes the compiler to get a stack dump

3600 Local BSS symbols mislocated (I1997)

4868 fatal 6 in assembler when -g used

5632 mips64-elf-gcc without -g on .S file does an improper instr reorder

5758 i960-vxworks-gas generating bad rellocation information

5784 .mdebug stabn local symbols have wrong layout

6326 Can not assemble certain MIPS machine instructions

6363 apparent bad output for weak symbol definition

6676 Delay slots not optimized correctly under specific circumstances.

6732 GAS-SH issues no error; leaves 0 length file as output

Assembler Preprocessor

6780 GASP changes a .data to a .long?

Compiler

1401 crts.o shouldn't be included in link for OSE targets.

1402 SOFT FP for 68k missinf from OSE release

1763 alignment of gcc structs

2068 question about register variables in gcc

2443 internel compiler error

2599 inline expansion of shift operations for long long desired

2643 i386-aout-gcc forgot about the "_" prefix for externals

2669 -m68020 options runs cpp with -Dmc68000

2718 Masking of unsigned int in big-endian region incorrect (#I779)

2941 gcc m68k optimizer oddity

3087 performance analysis: can we use gcc -a option

3841 warning: variable 'r' may be clobbered by 'longjmp'

4401 mips64-elf-gcc -pipe disabled in gcc specs file

4453 static globals allocated with incorrect size

4477 Can protoize and unprotoize be included in binary distribution?

4500 G++ passes parameters to function incorrectly

4580 stdio.h is not correctly fixed for AIX3.2.5 release

4593 compiler aborts on bitfield return

4600 Conflicts between cygnus and vxworks system header files & libraries

4613 libgcc.a for -m68000 has 68020 instructions (bsrl)

4713 inconsistent handling of #endif un-terminated comments

4744 Request more info on proposed changes in m68k-gcc

4841 lhu used instead of lbu for struct element access

4877 gcov directory handling problems

4894 gnu linker is not able to find libX11.a and won't link all

4905 gcc inline function has poor switch optimization on literals

5303 Help on gcc for SOLARIS

5304 Need option to say "all longs are aligned on 16 bit boundaries".

5403 gcc does incorrect code generation

5435 including math-68881.h emits code for atan, exp, log, and sqrt

5498

5521 compilation time seems excessive

5523 __extenddfxf2 not implemented for 68k -msoft-float

5524 compiler generates label LBB$0002 which gdb uses

5527 LynxOS linkage problems

5573 Internal compile error generated.

5662 Pointer manipulations generate bus errors

5702 bad .rodata and .data sections

5783 stabn's with out of order line numbers

5790 gcc infinite loop on bad C source

5849 Want option that behaves like GCC_EXEC_PREFIX (was: 94q4 distribution errors for mips64el-elf)

5913 -m68332 links with wrong libs

5951 gdb fails to recognize a struct type

5985 Debugger cannot show aggregate type

6022 Test of send-pr please ack.

6028 division by a constant is compiled incorrectly for R4000

6051 gcc has internal error with -O2 and -O3

6098 MIPS GCC code gen question

6138 Single precision floating point multiplication in mips-elf-940504 generates a FP exception.

6191 code generation for ternary expressions with write only memory fails

6342 new feature: -fpack-struct

6460 LISTING /home/ed/tmp/cca19507.s

6480 pointer compares on hppa not always handled correctly

6693 Invalid fixups in collector

6857 gcc on hppa generating bad code with -O[123]

7107 HPPA: function ptr compare crashes optimizer
page 1

6474 gcc generating calls to "sqrt"

6489 gcc should define__SVR4

6524 low nibble of int not stuffed into 12 bit bit field

6541 g++ finds incorrect include file

6546 invoking another assembler

6549 Re: Internal compiler error: program cc1 got fatal signal 11

6597 Re: GCC-SH: register unnecessarily allocated in gen_shifty_op?

6598 GCC-SH: andcosts() calculates costs incorrectly

6599 GCC-SH: REGISTER_MOVE_COST is too expensive for T-bit move

6603 Re: GNU mixed language programming

6623 GCC-SH: sh.md doesn't know ___ashrsi3 destroys r1

6655 Can gcc produce executables that are debugable by both gdb and dbx?

6672 Segmentation violation using strtok.

6675 gcc generates unaligned data access exception.

6731 gcc doesn't search /usr/local/lib/gcc-lib as claimed by docs

6768 mips64-elf jal delay slot filled with 'lw $2,value' is broken

6771 Warning: Can't load Codeset file 'C', using internal fallback

6775

Debugger

1686 gdb misdisplays some information

2006 gdb can't find a C++ fn that is not a member of a class

2072 gdb fails to find symbols

2085 gdb and stepping into "new"

2317 gdb fails to disassemble m68k "trap" instruction

2440 gdb internal error for i386-aout-gdb

2819 gdb cannot print C++ compiled structures defined without tags

2854 gdb does not properly handle cfront 3.0 names

3141 gdb: xcoff internal: pending include file exists.

3905 GDB unable to de-mangle template name (I2429)

3906 Backtrace loses info after superclass constructor call (I2430)

4000 segmentation violation when setting breakpoints. (I2934)

4179 Bus timeout too short (I3242)

4261 m88110-*gdb unable to find functions and local variables

4262 'next' and 'step' commands do not work as specified (I3358)

4279 m88110-bug-coff-objdump,gdb disassembly of trap instructions

4478 gdb is confused by external labels added to the code. (I4075)

4494 I get "badly mangled named" when I ptype *this

4556 gdb print elements cannot print 0 elements

4576 With language chill identifiers starting with '_' are not allowed

4610 gdb crashes within emacs-19

4903 gdb doesn't work correctly

5005 hppa backtrace consistently produces error message

5207 gdb can't properly debug c++ objects

5948 GDB question

5988 With --annotate=2, gdb dumps core on command sequences

6078 gdb new feature request in exception handling

6175 does not display correctly external var of type ptr to char

6302 gdb can't find data

6409 gdb cannot find source code on breakpoint

6443 Problems printing "this" in template function.

6535 gdb fails in trying to retrieve saved registers for a frame

6548 Issuing the "handle all print" command to gdb will generate a segmentation violation

6577 problem with invoking gdb in progressive-95q1

6591 gdb shared library version warnings

6592 gdb has some hardcoded pathnames

6642 gdb always starts windows?

6681 problem in remote debugging of m68k target using 95q1 gdb

6764 GDB GUI Help missing, would like to run non-GUI

Installation

6566 installation problems with SunOS 4.1.3 / Solaris 2.3

6640 Installation error Cygnus Support Developer's Kit

6644 Install is broken on OSF1 2.0

6765 Install script doesn't recognise OSF/1 2.0

Linker

2330 ld places wrong "machine type" into A.OUT output files

4785 ld ignores -T option

4788 ld generates errors for variables assignments over 16 bits

4803 ld fails with segmentation violation

5136 gnu ld with g++ on alpha-osf1.3 does not generate viable executables

5312 "ld" for AMD29K consumes excessive resources

5699 Use of DEFINED() function in scripts causes core dump.

5751 data section items not aligned correctly with -G0 option

5910 problem with R4000 compiler

6049 Do we support runtime loadable libraries (dlopen) on hppa?

6157 Don't understand output formats and Endian issues

6170 loader coredumps w/o error on object files w/different versions

6238 weak symbols defined first should be overidden

6239 94q4 mips64-elf-ld abort() in map_program_segments()

6258 ld -O a.out-mach3 does not produce a.out output

6341 Can not load output from the linker: "Backward seek"

6343 bad relocation

6397 ld aborts

6414 ld crashes when cross-compiling for the hurd on netbsd

6450 Link on Solaris platform fails with "bfd assertion fail"

6476 ld -shared sets execute bit on output file

6637 Linker gets memory exhausted error

6678 -split-by-reloc does not consider alignment characteristics

6791 code that linked with old version of ld now has undefined references

6796 abort during linking

6853 Linker doesn't order text as requested on command line

6893 Can't link standard library functions

6969 AT keyword in linker script does not work.

6998 File object produce errors while the ld command

6999 File object produce errors while the ld command

7024 linker reports "No symbols" in library

7025 linker is running out of memory

7063 ld of common symbols possibly not correct

7080 LD-SH: LD incorrectly resolves label

C Support Library

5202 statics remain in libc which should be in reent

5241 memory leaks from Balloc

6968 Which libc to use for 68360 compiles?

C++ Support Library

1244 <<long long can call <<char, conversion and

1245 Does the g++ library restriction apply to embedded medical systems?

2254 question on newer vintage of g++

2269 builtin.h max doesn't work

2824 warning: use of '...' without a first argument is non-portable

3660 string.h does not declare bfill(). (I2121)

4242 progessive-930929 has stopped work

4463 iomanip.h needs changes to work-around some defects

5209 iostream gives erroneous value for INT_MIN

6726 large # of I/O operations in program vs. older GNU version

Make

2017 make dependencies using archive files have a problem

2372 verbose messages from recursive make considered a defect

2441 make long-named options broken

4416 'make' does not run under solaris2.2

4676 How to include files conditionally depending on target without recursion

6645 can't override make variable with trailing + character in name

Motorola Assembler

2157 Difference between mas and Motorola compatible assembler

C and Math Support Libraries

2542	newlib problem
2735	gcvt does not work for negative numbers
2852	order of #includes can hide wchar_t (and others)
3285	Line buffered files have problems and lose data
3286	printf("%s",NULL) prints "null" rather than ""
3287	localtime returns pointer to a dynamic variable
3495	C library context structure too big
4053	localtime and mktime functions do no handle leap years right
4417	libc 'System Calls' documentation missing 'open()'
4583	Trouble finding library routines
4594	cannot include both math.h and math-68881.h
4733	Balloc should use calloc ?
4960	embedded errno facility is not extensible
5434	cannot include both math.h and math-68881.h
5531	Bus error caused by improper use of Balloc in dtoa.c
5547	toupper/tolower reference argument twice
6497	stdio and printf
6500	sprintf fails on some numbers
6595	atof("0.123456789012345678") -> infinite loop
6702	missing multibyte character functions
6708	pointer returned by [cm]alloc not aligned properly

Problem Reporting

4941	make install puts send-pr in wrong place
5364	send-pr.el gives bogus default response
5438	send-pr fails due to shell-command-on-buffer and point

Appendix A Graphical User Interface for GDB

GDB in this release is configured to use the new Graphical User Interface.

To run the GUI under Unix, just type 'gdb'. If your windowing environment is set up correctly (see "Getting Started," page 33) the GUI pops up automatically. (To run GDB without the GUI, use 'gdb -nw'.)

The GUI is based on Tcl and the Tk windowing toolkit; a different system was used to develop the GUI under Windows (available soon), but the underlying structure is the same.

When running as a Unix program and using the X11-based interface, you must of course be using an X server and/or workstation and your DISPLAY environment variable must be set correctly. If either of these is not true, then GDB starts up using the traditional command interface.

The exact layout and appearance of the windows depends upon the host system type. General behavior and layout is consistent across all platforms; omissions or restrictions on particular platforms, if not documented as unavoidable, should be considered bugs and reported.

All GDB windows have a common structure. Each window has an associated menu bar, which may be at the top of the window or perhaps elsewhere. Some of the menus and menu items in the menu bar are common to all GDB windows, while others are specific to particular types of windows. Below the menu bar is the working data area of the window. If the data is too large to display all at once, the data area shows scroll bars on its right and bottom sides. Below the data area are two optional features: a status/data line, and a button box.

For details on using GDB, see section "GDB Commands" in *Debugging With GDB*.

Getting Started

To launch the GDB GUI, simply type gdb on the command line. If you are in a windowing environment and your DISPLAY variable is set correctly, the Command Window and Source Window appear. To suppress the GUI, use the '-nw' (*non-windowing*) option on the command line.

If you move the binaries to somewhere other than their original location, you need to set the following three environment variables:

```
TCL_LIBRARY
TK_LIBRARY
GDBTK_FILENAME
```

TCL_LIBRARY is the location (directory) of tcl code that tcl expects to be able to find, while TK_LIBRARY is the same for tk.

GDBTK_FILENAME is the pathname to the tcl code that actually defines the GDB graphical interface.

For example, to set these environment variables if you installed in the standard location for this release ('/usr/cygnus/progressive-95q3/...'), use the following (backslashes, \, indicate a line continuation even though a line break is shown; this is to make the example able to fit on the printed page).

```
set TCL_LIBRARY=\
  /usr/cygnus/progressive-95q3/lib/tcl
export TCL_LIBRARY
set TK_LIBRARY=\
  /usr/cygnus/progressive-95q3/lib/tk
export TK_LIBRARY
set GDBTK_FILENAME=\
  /usr/cygnus/progressive-95q3/H-hosttype/lib/gdbtk.tcl
export GDBTK_FILENAME
```

Note: sh syntax is shown. Use the syntax appropriate for your shell.

Menus

File Menu

The standard file menu provides operations that affect the overall state of GDB. These are mainly file operations, but other options exist as well.

File... Lets you set the combined executable and symbol file that GDB will use. (Like the gdb command 'file'.)

Target... Brings up a dialog that you can use to connect GDB to a target program. The dialog is described in more depth later. (Like the gdb command 'target'.)

Edit... Starts up an editor to modify the source file being displayed.

Exec File...
Lets you set the executable file that GDB will use. (Like the gdb command 'exec-file'.)

Symbol File...
Lets you set the symbol file that GDB will use. (Like the gdb command 'symbol-file'.)

Add Symbol File...
Lets you add additional symbol files. (Like the `gdb` command '`add-symbol-file`'.)

Core File...
Lets you set the core file that GDB will use. (Like the `gdb` command '`core-file`'.)

Shared Libraries...
(Like the `gdb` command '`sharedlibrary`'.)

Quit Quits GDB. (Like the `gdb` command '`quit`'.)

Options Menu
The Options Menu is different for each window, showing the viewing options available within that window. In the Source Window, for example, one of the options is whether to display line numbers.

Window Menu
The Window Menu allows access to all the windows available in GDB. The first part of the menu lists all of the predefined individual windows. If the window exists already, its item will be marked as such; selecting the item will cause the window to be put in front if it is obscured. If the window does not exist, then it will be created.

The second part of the menu lists additional windows that you may have created, such as source windows or variable displays.

Selections on this menu include:

- Command
- Source
- Assembly
- Registers
- Variables
- Files
- *any other windows you have open*

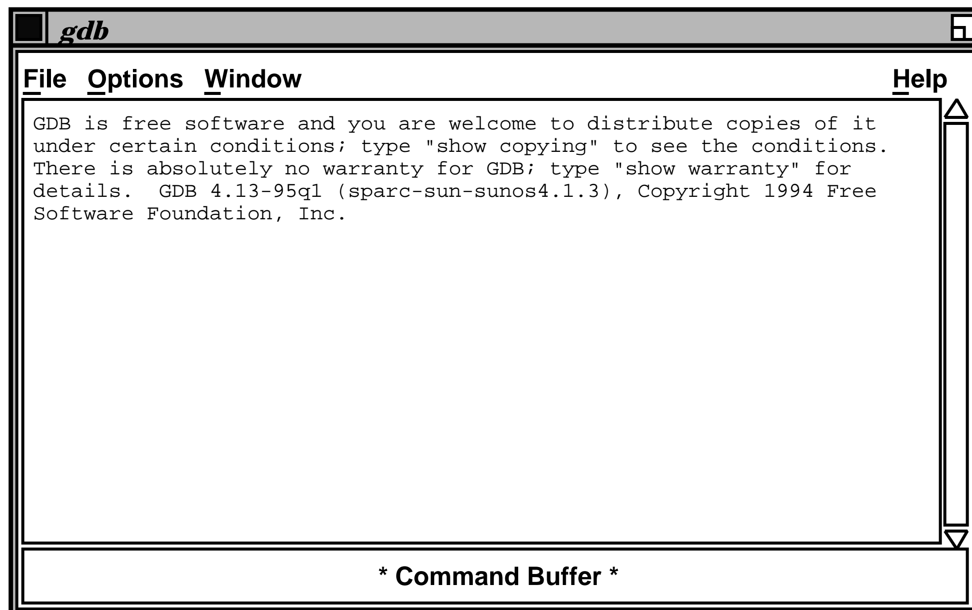
Help Menu
The Help Menu includes access to GDB's online help.

Windows

Command Window

The command window provides access to the standard GDB command interpreter. In nearly all cases, commands typed into this window behave exactly as for a non-windowing GDB.

Note that not all changes to GDB are be reflected in this window. For instance, if you type a '`step`' command, then click on the '`step`' menu item in the source window, then type another '`step`' command, the command buffer shows only two steps, although you have actually done three. GDB places an ellipsis (...) in the command buffer when operations in other windows are done, as a reminder that the command buffer is incomplete. The command window has no status line or button box.



(This example screen is the initial Command Window from the Unix version of the GDB GUI*.)*

Files Window

The Files Window lists all of the files that were used to build the executable.

Clicking in the bar in the left margin expands/contracts the display of included files and symbols defined by the file.

Source Window

A source window displays a single file of source code.

The left margin includes an indicator for the current PC, breakpoints and potential breakpoints, and (optionally) line numbers.

Appendix B Specifying Names for Hosts and Targets

Your tape is labeled to indicate the host (and target, if applicable) for which the binaries in the distribution are configured. The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, though the scheme is slightly different between hosts and targets.

Host names

The full naming scheme for hosts encodes three pieces of information in the following pattern:

architecture-vendor-os

For example, the full name for a Sun SPARCstation running SunOS 4.1.3 is

```
sparc-sun-sunos4.1.3
```

> *Warning:* `configure` can represent a very large number of combinations of architecture, vendor, and operating system. There is by no means support for all possible combinations!

The following combinations refer to hosts supported by Cygnus Support. Some common short aliases are included, but these may be obsolete in the future. (For a matrix which shows all supported host/target combinations, see section "Overview" in *Release Notes*.)

canonical name	*alias*	*platform*
`sparc-sun-solaris2`	`sun4sol2`	Sun 4 running Solaris 2
`sparc-sun-sunos4.1.3`	`sun4`	Sun-4 running SunOS 4
`mips-dec-ultrix`	`decstation`	DECstation
`rs6000-ibm-aix`	`rs6000`	IBM RS6000
`mips-sgi-irix4`	`iris`	SGI Iris running Irix 4
`m68k-hp-hpux`	`hp300hpux`	HP 9000/300
`hppa1.1-hp-hpux`	`hp700`	HP 9000/700
`i386-unknown-sysv4`		UnixWare
`alpha-dec-osf1.3`		DEC Alpha running OSF/1 v1.3

Target names

If you have a cross-development tape, the label also indicates the target for that configuration. The pattern for target names is

architecture[-*vendor*]-*objfmt*

Target names differ slightly from host names in that the last variable indicates the object format rather than the operating system, and the

second variable is often left out (this practice is becoming obsolete; in the future, all configuration names will be made up of three parts).

In cross-development configurations, each tool in the Developer's Kit is installed with the configured name of the target as a prefix. For example, if the C compiler is configured to generate COFF format code for the Motorola 680x0 family, the compiler is installed as '`m68k-coff-gcc`'.

> *Warning:* `configure` can represent a very large number of target name combinations of architecture, vendor, and object format. There is by no means support for all possible combinations!

This is a list of some of the more common targets supported by Cygnus Support. (Not all targets are supported on every host!) The list is not exhaustive; see section "Overview" in *Release Notes*, for an up-to-date matrix which shows the host/target combinations supported by Cygnus.

Motorola 68000 family

`m68k-aout`	`a.out` object code format
`m68k-coff`	COFF object code format
`m68k-vxworks`	VxWorks environment

Motorola 88000 family

`m88k-coff`	COFF object code format

Intel 960 family

`i960-vxworks5.0`	VxWorks environment (`b.out` format)
`i960-vxworks5.1`	VxWorks environment (COFF format
`i960-intel-nindy`	Nindy monitor

AMD 29000 family

`a29k-amd-udi`	UDI monitor interface

To use the minimon *interface, use this configuration with the auxiliary program* MONTIP*, available from AMD.*

SPARC family

`sparc-vxworks`	VxWorks environment
`sparc-aout`	`a.out` object code format
`sparclite-aout`	`a.out` object code format
`sparclite-coff`	COFF object code format

Intel 80X86 family

`i386-aout`	`a.out` object code format
`i386-netware`	NetWare NLM

IDT/MIPS R3000

`mips-idt-ecoff`	IDT R3000, big endian ECOFF
`mipsel-idt-ecoff`	IDT R3000, little endian ECOFF
`mips64-idt-ecoff`	IDT R4000 ECOFF

Hitachi H8300

`h8300-hms-coff`	COFF object code format

Hitachi SH

`sh-hms-coff`	COFF object code format

z8000

`z8k-coff`	COFF object code format

`config.guess`

`config.guess` is a shell script which attempts to deduce the host type from which it is called, using system commands like `uname` if they are available. `config.guess` is remarkably adept at deciphering the proper configuration for your host; if you are building a tree to run on the same host on which you're building it, we recommend *not* specifying the *hosttype* argument.

`config.guess` is called by `configure`; you need never run it by hand, unless you're curious about the output.

Installation Notes

Cygnus Support Developer's Kit
Release `progressive-94q4`

Contents:

Brief installation instructions:
Installing in brief for Unix systems, page 1.
Installing in brief for MS-DOS systems, page 3.

Detailed installation information:
Developer's Kit installation on Unix, page 5.
Developer's Kit installation on MS-DOS, page 21.

Appendices:
Appendix A Platform names, page 27.
Appendix B Cross-development environment, page 31.

Cygnus Support hotline: +1 415 903 1401

Copyright © 1994, 1995 Cygnus Support

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

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

Installing in brief for Unix systems

You can run the brief installation procedure if:

- You are installing on a standard Unix platform (see Appendix A "Platform Names," page 27)
- Your Unix machine has its own device which corresponds to your distribution media (e.g., a QIC-24 tape drive)
- You're willing to use the installation directory '`/usr/cygnus`'
- You have enough disk space in '`/usr/cygnus`' (your tape label lists the required disk space for binary, source, and both)

Otherwise, see "Developer's Kit installation on Unix," page 7.

Steps for Brief Install:

(In examples, we show the system prompt as '`eg$`'.)

1. Make sure you can write in '`/usr/cygnus`' by typing:

```
eg$ su root                  (enter root password)
# mkdir /usr/cygnus          (ignore "File exists" error if any)
# chmod 777 /usr/cygnus
# exit                       (root access not needed beyond this)
```

2. Load the distribution into the drive and extract the `Install` script. **WARNING:** you must use a *non-rewinding* tape device; see "Device names," page 5.

```
eg$ cd /tmp
eg$ tar xfv device Install
```

3. Run the `Install` script:

```
eg$ ./Install -tape=device
```

`Install` displays messages about its activity, ending with

```
Done.
```

4. Build symbolic links to make execution paths easy (you may need root access to put the link in '`/usr`'):

```
eg$ cd /usr/cygnus
eg$ ln -s progressive-94q4 progressive
eg$ su root
# ln -s /usr/cygnus/progressive/H-host /usr/progressive
# exit                  (give up root access as soon as possible)
```

5. Use your Cygnus Support customer ID (see cover letter) to tag your copy of our problem-report form:

```
eg$ /usr/progressive/bin/install-sid customer-ID
```

6. Remove public write access from '`/usr/cygnus`'. See your system administrator for the correct permissions at your site.

You're done! Anyone who puts '`/usr/progressive/bin`' in their `PATH` can use this Developer's Kit distribution.

We have attempted to make the installation of the Cygnus Support Developer's Kit distribution as trouble-free as possible. If you encounter any problems, please contact us:

Cygnus Support
toll free: +1 800 CYGNUS-1
main line: +1 415 903 1400
hotline: +1 415 903 1401
email: `support@cygnus.com`

Headquarters
1937 Landings Drive
Mountain View, CA 94043 USA

+1 415 903 1400
+1 415 903 0122 fax

East Coast
48 Grove St., Ste. 105
Somerville, MA 02144 USA

+1 617 629 3000
fax +1 617 629 3010

Faxes are answered 8 am–5pm, Monday through Friday.

Installing in brief for MS-DOS systems

All MS-DOS releases of the Developer's Kit can use this brief installation procedure. For more detail on the installation procedure, see "Developer's Kit installation on MS-DOS," page 21.

We ship your Developer's Kit on a set of floppy disks. The `INSTALL` program is included on Disk 1.

Steps for Brief Install:

1. Insert Disk 1 into the floppy drive (the example shows '`A:`') and type:

   ```
   A:\INSTALL
   ```

2. `INSTALL` prompts for an installation directory; the default is '`C:\CYGNUS`'. The tools may be installed anywhere. (If you are installing more than one Developer's Kit, see "MS-DOS Installation Directories," page 21.)

   Make sure the installation directory is correct and press `RETURN`.

3. `INSTALL` first checks the installation location to make sure it has enough space before unpacking the tools. Installed Developer's Kit disk usage varies from about 10 to about 16 megabytes, depending on the target. The disk labels list the required disk space for each distribution.

4. `INSTALL` reads the first disk and then requests the next.

5. After the last disk, press `RETURN` to exit the `INSTALL` program.

6. To run the programs, type '*installdir*\SETENV' to set up your working environment. (*installdir* is the installation directory you specified, '`C:\CYGNUS`' by default.)

   ```
   C:\> cd c:\cygnus
   C:\CYGNUS\> setenv
   ...
   ```

7. To test the installation, change your working directory to the '*installdir*\DEMO' subdirectory and type '`MAKE`':

   ```
   C:\CYGNUS\> cd c:\cygnus\demo
   C:\CYGNUS\DEMO> make
   ...
   ```

8. It is often easiest to set your working environment in the initialization file '`AUTOEXEC.BAT`'. If you installed in the default '`C:\CYGNUS`', you can simply add the following line to your '`AUTOEXEC.BAT`':

   ```
   CALL C:\CYGNUS\SETENV.BAT
   ```

Developer’s Kit installation on Unix

This section describes host-specific information and installation instructions for Unix systems.

Device names

For Unix distributions, your distribution tape includes two files:

`Install` The `Install` shell script is a portable installation procedure which automatically installs the software on your system (see “Invoking the `Install` script,” page 8).

distribution `tar` *file*
The binaries and source for the Developer’s Kit distribution are located in a single compressed `tar` file.

> In order for `Install` to read and install your distribution, you *must* use a *non–rewinding* tape device, so that the tape drive maintains the tape location at the beginning of the compressed `tar` file after you extract `Install`.

These are examples of non–rewinding tape devices for each system; see your system administrator for the name of the non–rewinding tape device on your particular host. Also shown are MAN pages to which you can refer for more information about tape devices.

Standard QIC-24 tape drives:

platform	*device*	*man page*
`sparc-sun-solaris2`	`/dev/rmt/0n`	*use* `man st`
`sparc-sun-sunos4.1.3`	`/dev/nrst8`	*use* `man st`
`mips-dec-ultrix`	`/dev/nrtm0`	*use* `man mtio`
`mips-sgi-irix4`	`/dev/mt/tps0d0nrns`	*use* `man tps`
(Note: You must also use a non-byte-swapping device for the SGI Iris)		
`rs6000-ibm-aix`	`/dev/rmt0.1`	*use* `man rmt`
`i386-sysv4.2`	`/dev/rmt/c0s0n`	*use* `man tape`

Standard DAT tape drives:

platform	*device*	*man page*
`m68k-hp-hpux`	`/dev/rmt/0mn`	*use* `man 7 mt`
`hppa1.1-hp-hpux`	`/dev/rmt/0mn`	*use* `man 7 mt`
`mips-sgi-irix5`	`/dev/mt/tps1d5nrns`	*use* `man tps`
(Note: You must also use a non-byte-swapping device for the SGI Iris)		

Disk space requirements

This section lists the minimum disk space requirements (in megabytes) for installations of binaries only, source code only, or the sum total of both. For more information on binary- or source-only installations, see "Developer's Kit installation on Unix," page 7. Sizes shown are for *native* distributions; see your tape label for the actual disk size.

platform	*binaries*	*source*	*total*
`sparc-sun-solaris2`	44	71	115
`sparc-sun-sunos4.1.3`	31	71	102
`mips-dec-ultrix`	39	71	110
`rs6000-ibm-aix`	44	71	115
`mips-sgi-irix4`	39	71	110
`m68k-hp-hpux`	32	71	103
`hppa1.1-hp-hpux`	46	71	117
`i386-sysv4.2`	48	71	119
`alpha-dec-osf1.3`	56	71	127

Operating System requirements

This section lists the minimal operating system requirements for each Unix system.

platform	*OS level*
`sparc-sun-solaris2`	Solaris 2.*x*
`sparc-sun-sunos4.1.3`	SunOS 4.1.*x*
`mips-dec-ultrix`	Ultrix 4.2
`rs6000-ibm-aix`	AIX 3.2
`mips-sgi-irix4`	Irix 4.*x*
`m68k-hp-hpux`	HP/UX 8.*x*
`hppa1.1-hp-hpux`	HP/UX 8.*x*
`i386-sysv4.2`	UnixWare SysVr4.2, version 1.1.1
`alpha-dec-osf1.3`	OSF/1 1.3

Installing your Developer's Kit distribution

There are a few steps to follow in installing the software in the Developer's Kit distribution onto your system.

Note: For Unix distributions, your distribution tape includes two files:

`Install` — The `Install` shell script is a portable installation procedure which automatically installs the software on your system (see "Invoking the `Install` script," page 8).

distribution `tar` *file* — The binaries and source for the Developer's Kit distribution are located in a single compressed `tar` file.

In order for `Install` to read and install your distribution, you *must* use a *non–rewinding* tape device, so that the tape drive maintains the tape location at the beginning of the compressed `tar` file after you extract `Install`. See "Device names," page 5, for a list of default device names for each host type.

1. First, decide where to install the software. The default installation location is '`/usr/cygnus/progressive-94q4`'. (To use the software conveniently from elsewhere, you may want to reconfigure and recompile from source; see "Running the programs," page 15.)

   If you don't wish to install in '`/usr/cygnus`' but can create a symbolic link to it from another location, or if you don't wish to install into '`/usr`' at all, see "Installing in a nonstandard location," page 14.

2. Create the installation directory, if it doesn't already exist, and make sure it's publicly accessible so `Install` can write there. For example, if you use the default installation directory of '`/usr/cygnus`':

```
eg$ su root                (enter root password to write in '/usr')
# mkdir /usr/cygnus        (ignore "File exists" error if any)
# chmod 777 /usr/cygnus
# exit                     (root access not needed beyond this)
```

3. Make sure you have enough space for the tools in your chosen installation location. The required disk usage for the Developer's Kit is printed on the tape label; values for binaries only, sources only, or both are shown.

4. Load the Developer's Kit distribution tape into your tape drive. If your machine doesn't have its own tape drive, you need to first extract the software into a location accessible by both your host and the machine that has a tape drive, and then install on your host. If there is no shared disk, you can extract the software on the machine with the tape drive and then transfer it over to your host. "Installing with a remote tape drive," page 14, for details.

5. Extract the `Install` script off the tape using

```
tar xvf non-rewinding-tapedev Install
```

Remember to use a *non–rewinding* tape drive!

6. Run `Install`, using command-line options and arguments to specify the details about your installation.

   Default behavior installs both binaries and source under '`/usr/cygnus/progressive-94q4`' using the default non–rewinding tape drive for your system (see "Device names," page 5). For *native* toolchains only, a process called *fixincludes* automatically makes copies of your system header files and alters them to work with GCC (your system's header files are *not changed*; see "Why convert system header files?," page 11). Finally, `Install` runs a simple test to make sure your distribution was installed correctly.

7. Make sure the program `send-pr` knows your Cygnus customer identification code. You can install your customer ID by using the program `install-sid` as follows:

   ```
   eg$ cd /usr/cygnus/progressive-94q4/H-hosttype/bin
   eg$ install-sid customer-ID
   ```

   Contact Cygnus Support at +1 415 903 1401 if you do not know your customer ID.

8. Create symbolic links so that your newly installed Developer's Kit is easily accessible to developers, able to exist with other Developer's Kit installations in a heterogeneous environment, and easily updated when you install a new Developer's Kit.

   The nature of the links depends on where you installed the Developer's Kit release, but they follow the example below. If you installed into '`/usr/cygnus/progressive-94q4`', the links are

   ```
   ln -s /usr/cygnus/progressive-94q4 /usr/cygnus/progressive
   ln -s /usr/cygnus/progressive/H-hosttype /usr/progressive
   ```

   See "Links for easy access and updating," page 12, for more information on these links.

   You're done! The installation is now online; anyone who puts

   ```
   /usr/progressive/bin
   ```

   in their path has access to the toolkit.

9. If you had to change the permission status on the directory '`/usr/cygnus`', be sure to revert the change. See your system administrator for the proper permissions at your site.

Invoking the `Install` script

There are two kinds of command-line arguments to `Install`, which you can use to direct its operation:

- *What form of the programs* to install. You can choose between binaries (argument `bin`) and source code (`source`). If you don't specify either of these, `Install` assumes you want *both source and binaries*.
- *What installation actions* to carry out. A full installation involves up to three steps; `Install` has options to let you choose them explicitly. The steps are
  1. extracting source from the tape (option `extract`)
  2. writing copies of your system '`include`' files, adjusted for portability (needed for the compilation tools; option `fixincludes`)
  3. running a simple test of the installed programs (option `test`)

  The last two of these actions (`fixincludes` and `test`) are *not needed for cross-development* configurations. (A cross-development configuration runs on a *host*, but is meant to develop code for another platform, the *target*. Cross development tapes have '`target = target`' on the tape label.)

  These two actions can only run on your host. If you read the tape on another machine, you must specify the `extract` option explicitly, to indicate that you don't expect the other two actions to run (and are aware of the need to run further installation steps on your host).

`Install` also has two command line options: '`-tape`' and '`-installdir`'. You can use these to adapt the installation to your system.

`Install` options

```
Install [ bin ] [ source ]
        [ extract ] [ fixincludes ] [ test ]
        [ -tape=device ]
        [ -installdir=directory ]
```

`bin`
`source`
: By default, `Install` extracts both source and binaries. Instead of relying on the default, you can use these options to specify exactly what you want. You need to do this if you want *only* binaries or *only* source.

  `Install` is designed to share files, wherever possible, between installations for different hosts (of the same release). If you get Cygnus release tapes configured for different hosts, there is no need to do a binary-only install of some of the tapes to save space on a shared file system; `Install` arranges the files so that all hosts share the same source files. Documentation files are shared as well. Note that it is faster to extract

the source code only once if you are installing the Developer's Kit distribution for more than one host.

See "Links for easy access and updating," page 12, for a discussion of how to manage the directory structure used for this purpose.

`extract`
`fixincludes`
`test`

In a cross-development configuration, only the '`extract`' step is used.

In a native configuration—meant for developing software on the same host where the Developer's Kit runs—a full installation includes up to three things: (1) extracting software from the tape; (2) creating ANSI-C conforming copies of your system's standard header files; and (3) testing the installation. You can execute these steps separately by specifying '`extract`', '`fixincludes`', or '`test`' on the `Install` command line.

In the native configuration, after you run '`extract`', '`fixincludes`' is essential to the compiler. '`fixincludes`' *does not change your system's original header files;* `Install` writes the converted copies in a separate, `gcc`-specific directory. See "Why convert system header files?," page 11, for more discussion of the '`fixincludes`' step. `Install` only attempts these last two steps if you run it on the host for which the binaries were compiled.

When you run '`extract`', `Install` creates a log file in '`/usr/cygnus/progressive-94q4/extraction.log`'. When you run '`fixincludes`', `Install` creates a log file in '`/usr/cygnus/progressive-94q4/fixincludes.log`'.

'`test`' (used only for the native configuration) is a confidence-building step, and doesn't actually change the state of the installed software. The '`test`' step may not make sense, depending on what other options you've specified—if you install only source, there's nothing to test.

`-tape=`*`device`*
`-tape=`*`tarfile`*

Specify the *non–rewinding* device name for your tape drive as *`tape`*.

If you extract the installation script and *`tarfile`* on some other system, and transfer them to your host for installation, use the name of the `tar` file instead of a device name with

'-tape'. See "Installing with a remote tape drive," page 14, for more discussion.

-installdir=*directory*
: If you cannot or do not wish to install into '/usr/cygnus', use this option to specify an alternate *directory* for placing your software—but beware: the software is configured to go in '/usr/cygnus', and you'll have to override or change that too. See "Running the programs," page 15.

If you specify a step that doesn't make sense, Install notices the error, and exits (before doing anything at all) with an error message, so you can try again.

Why convert system header files?

> The 'fixincludes' installation step described here *applies only to the native configuration* of the Developer's Kit—that is, only if your tape is configured to develop software for the same *host* on which it runs. If you have a cross-development tape, configured to develop software for another machine (the *target*), the system header files from your *host* are not needed for the GNU compilers. Cross-development tapes have 'target = *target*' on the tape label.

For the native configuration, it is very important to run 'Install fixincludes' (on *each host* where you install the compiler binaries).

When the ANSI X3J11 committee finished developing a standard for the C language, a few things that had worked one way in many traditional C compilers ended up working differently in ANSI C. Most of these changes are improvements. But some Unix header files still rely on the old C meanings, in cases where the Unix vendor has not yet converted to using an ANSI C compiler for the operating system itself. 'Install fixincludes' does a mechanical translation that writes ANSI C versions of some system header files into a new, GCC-specific include directory—*your system's original header files are not affected.*

The C header files supplied with SVr4 versions of Unix depend on a questionable interpretation of the ANSI C standard: they test for a non-ANSI environment by checking whether __STDC__ is defined as zero. The ANSI standard actually only specifies that __STDC__ be defined to 1; if it is defined to any other value, the environment is not ANSI C compatible, and ANSI C says nothing about what that value might be.

GCC defines __STDC__ to 1 when running with '-ansi', when it functions as an "ANSI C superset" compiler. (It also sets __STRICT_ANSI__

when it runs with the '-pedantic' option.) However, GCC leaves __STDC__ undefined when it is not running as an ANSI C compiler.

Unfortunately for Solaris users, Solaris header files follow the SVr4 choice. Since GCC never defines __STDC__ as 0, the distributed header files can leave out some declarations. (Look in '/usr/include/time.h', for example.)

Part of the installation process of the native compiler release is to "fix" the header files, such as 'stdio.h', on the host system to remove ANSI incompatibilities. 'Install fixincludes' makes copies of the system 'include' files which have these nonstandard features removed, so that GCC can process them. These copies are placed in a new, GCC-specific 'include' directory—*your system's original header files are not affected.* Once these fixed header files are created, GCC finds and uses them automatically.

Likewise, C++ programmers require C++-ready, ANSI-compatible versions of the standard C header files. These used to be provided with libg++, but were difficult to maintain due to the design compromises (and outright "kludges") that were necessary to make these work on all the systems we support.

We have recently introduced what we believe to be a better solution in the form of a new shell script, fixproto. fixproto analyzes all the header files in '/usr/include', and adds any missing standard ANSI and Posix.2 prototypes. The 'extern "C"' braces needed to specify that these are C (not C++) functions are also added as needed. It is run as part of the installation and/or build of a native compiler. The resulting header files are also used for C, with the result that the '-Wimplicit' option for gcc is much more useful.

The most obvious drawback to this solution is that the process of "fixing" the 'include' files takes longer to run, so any installation of a native compiler is noticeably slower than in previous releases. Performance improvements will be made as part of a future release.

If you don't run fixincludes, the GNU C compiler can only use the original system header files when you compile new C programs. *In some cases, the resulting programs will fail at run-time.*

Links for easy access and updating

Once you've extracted the tools from the tape, they are installed into a directory named '*installdir*/progressive-94q4'. We put the release number in the directory name so that you can keep several releases installed at the same time, if you wish. In order to simplify administrative procedures (such as upgrades to future Cygnus Support Pro-

gressive releases), we recommend that you establish a symbolic link '`/usr/cygnus/progressive`' to this directory.

```
ln -s installdir/progressive-94q4 installdir/progressive
```

For example, if you've installed in the default location under '`/usr/cygnus`':

```
ln -s /usr/cygnus/progressive-94q4 /usr/cygnus/progressive
```

Directories of *machine-independent* files (source code and documentation) are installed directly under '`progressive-94q4`'. However, to accomodate binaries for multiple hosts in a single directory structure, the binary files for your particular host type are in a subdirectory '`H-`*`hosttype`*'. (*`hosttype`* indicates a particular architecture, vendor and operating system. See Appendix A "Platform names," page 27.)

This means that one more level of symbolic links is helpful, to allow your users to keep the same execution path defined even if they sometimes use binaries for one machine and sometimes for another. Even if this doesn't apply now, you might want it in the future; establishing these links now can save your users the trouble of changing all their paths later. The idea is to build '`/usr/progressive/bin`' on each machine so that it points to the appropriate binary subdirectory for each machine—for instance, '`/usr/cygnus/progressive/H-`*`hosttype`*'.

You may need super-user access again briefly to establish this link:

```
ln -s /usr/cygnus/progressive/H-hosttype /usr/progressive
```

We recommend building these links as the last step in the installation process. That way, users at your site only see software in '`/usr/progressive`' when you're satisfied that the installation is complete and successful.

Installation variances

Once you've extracted `Install` from your tape, you can tell `Install` what software to install, what form of the programs you need, and what installation steps to do. Here are some examples covering common situations. For a full explanation of each possible `Install` argument, see "Invoking the `Install` script," page 8.

`Install`'s default tape drive is the non-rewinding tape drive for your system (see "Device names," page 5), which is right for the most common cases. If your tape drive is different, you need to use the '`-tape=/dev/`*`tape`*' option; the examples show this option for completeness. Remember to specify a *non–rewinding* tape device.

Installing only binaries or source

If you don't want the source—for instance, to save space—you can use the argument 'bin'.

```
eg$ tar xvf device Install
Install
eg$ ./Install -tape=device bin . . .
```

By the same token, if you don't wish to install the binaries—for instance, if you plan to rebuild them from source anyway—you can use the argument 'source'.

```
eg$ tar xvf device Install
Install
eg$ ./Install -tape=device source . . .
```

Installing in a nonstandard location

If you wish to install this Developer's Kit distribution in a directory other than the default, '/usr/cygnus', use the '-installdir' option to Install. Remember, though, you must set some environment variables in order for the tools to function at all. See "Running the programs," page 15.

```
eg$ cd /tmp
eg$ tar xvf device Install
Install
eg$ ./Install -tape=device -installdir=somewhere bin
. . .
```

Installing with a remote tape drive

If your host doesn't have an appropriate tape drive, you may still be able to install your software. Check with your system administrator to see if another machine at your site has a tape drive you can use. If so:

If a shared filesystem is available

between the two machines, and it has enough space, create '/usr/cygnus' on your host (the one where you want to install this Progressive Release) as a symbolic link to a directory where the other machine (the one with a tape drive) can write:

```
ln -s shared /usr/cygnus
```

Run Install from the machine with a tape drive, using the 'extract' argument and the '-installdir' option:

```
Install extract -installdir=shared
```

You still have to finish the installation, but the last two steps (fixincludes and test) must be run on your host. (If you

forget, there's no great harm done: `Install` notices that it can't carry out a full installation on the wrong machine, and stops with an error message—then you can go back and try again. When `Install` notices a problem like this, it doesn't carry out *any* action other than giving a helpful error message).

Unless you are installing a cross-development tape (the tape label says '`target = target`' for cross configurations), the '`fixincludes`' part of the installation is essential. Please see the full explanation (see "Why convert system header files?," page 11), if you're curious.

On a machine on your network with a tape drive:

```
./Install extract -installdir=shared/cygnus . . .
```

On your host

```
ln -s shared/cygnus /usr/cygnus
cd /usr/cygnus/progressive-94q4
```

If your copy of the Developer's Kit is configured *native* (to develop software for the same type of machine where the Developer's Kit itself runs), you'll have to run '`Install fixincludes`' and '`Install test`' from your host afterwards.

Native configurations only:

```
./Install fixincludes test
```

If some form of filetransfer is available

(such as `uucp`), read the tape using a system utility (for instance, `dd` on Unix systems; see the system documentation for the machine with a tape drive). There are two files on the distribution tape; the first contains just the `Install` script, and the second is a compressed `tar` format file containing the rest of the release. Read both of these files, and transfer them to your own machine. Then run `Install`, but use '`-tape=tarfile`' to specify the name of the installation file, instead of '`-tape=device`' as shown in the examples. In the simplest case, for example (starting after you've transferred `Install` and the tar file to your system):

```
eg$ ./Install -tape=tarfile
```

Running the programs

In order to run the tools in the Developer's Kit release after you install them, you must first set a few environment variables so your shell can find them.

- At the very least, you must set your `PATH` variable. See "Setting PATH," page 16.

- If you installed the tools in a location other than the default and choose not to set the standard symbolic links in place (see "Links for easy access and updating," page 12), you must also set the environment variable `GCC_EXEC_PREFIX`. Otherwise, the compiler cannot find its resources. See "GCC paths," page 16.
- If you install the Developer's Kit tools in an alternate location, you need to set the variable `INFOPATH` so that `info` can find the online documentation. See "Online documentation paths," page 17.
- Some `man` programs recognize the environment variable `MANPATH` as a search path for online manual pages. You must either add your installation directory to your `MANPATH` environment variable, or copy the online manual pages in your distribution into a location where your `man` program can find them. See "Online documentation paths," page 17.

Setting PATH

Any user who wishes to run the tools in this distribution needs to make sure her `PATH` environment variable can find the tools. Whether you install in the default location:

```
/usr/cygnus/progressive-94q4
```

or in an alternate location, you need to alter your `PATH` environment variable to point toward the newly installed tools.

If you create the symbolic links we recommend (see "Links for easy access and updating," page 12), users who want to run the Developer's Kit—regardless of whether they need binaries for your particular host, or for some other platform—can use settings like one of the following in their initialization files.

This example shows '`/usr/progressive/bin`' as the final linked installation directory. If you installed into a directory other than this, substitute the actual directory for '`/usr/progressive/bin`'.

For Bourne-compatible shells (`/bin/sh`*,* `bash`*, or Korn shell):*

```
PATH=/usr/progressive/bin:$PATH
export PATH
```

For C shell:

```
set path=(/usr/progressive/bin $path)
```

GCC paths

You can run the compiler GCC without recompiling, even if you install the distribution in an alternate location, by first setting the environment

variable `GCC_EXEC_PREFIX`. This variable specifies where to find the executables, libraries, and data files used by the compiler. Its value will be different depending on which set of binaries you need to run. For example, if you install the tape distribution under '`/local`' (instead of the default '`/usr/cygnus`'), and you wish to run GCC as a native compiler, you could set `GCC_EXEC_PREFIX` as follows.

(*Note:* The sample shows a `GCC_EXEC_PREFIX` which is split across two lines only to fit on the printed page; it is meant to be typed on one line.)

For shells compatible with Bourne shell (`/bin/sh`*,* `bash`*, or Korn shell):*

```
eg$ GCC_EXEC_PREFIX=/local/progressive-94q4/\
      H-hosttype/lib/gcc-lib/
eg$ export GCC_EXEC_PREFIX
```

For C shell:

```
eg% setenv GCC_EXEC_PREFIX /local/progressive-94q4/\
      H-hosttype/lib/gcc-lib/
```

Note: The trailing slash '`/`*' is important.* The `gcc` program uses `GCC_EXEC_PREFIX` simply as a prefix. If you omit the slash (or make any other mistakes in specifying the prefix), `gcc` fails with a message beginning '`installation problem, cannot exec...`'.

Online documentation paths

The standalone documentation browser `info` needs to know the location of the documentation files in the distribution. The default location, '`/usr/cygnus/progressive-94q4/info`', is compiled into `info`. If you install elsewhere, set the environment variable `INFOPATH` to indicate the alternate location.

For example, assuming you installed under '`/local`':

For shells compatible with Bourne shell (`/bin/sh`*,* `bash`*, or Korn shell):*

```
eg$ INFOPATH=/local/progressive-94q4/info
eg$ export INFOPATH
```

For C shell:

```
eg% setenv INFOPATH /local/progressive-94q4/info
```

If you built '`progressive`' as a symbolic link to '`progressive-94q4`', as recommended in "Links for easy access and updating," page 12, then you could simply use '`/local/progressive/info`' as the value of `INFOPATH` in the examples above.

You should also ensure that your `man` command can pick up the manual pages for these tools. Some `man` programs recognize a `MANPATH` environment variable. If your `man` program is one of these, users at your site can also include in their initialization file lines like

For Bourne-compatible shells:

```
eg$ MANPATH=/usr/cygnus/progressive/man:$MANPATH:/usr/man
eg$ export MANPATH
```

For C shell:

```
eg% setenv MANPATH /usr/cygnus/progressive/man:$MANPATH:/usr/man
```

If your `man` program doesn't recognize `MANPATH`, you may want to copy or link the files from '*installdir*/progressive/man/man1' into your system's 'man/man1' directory.

Some Things that Might go Wrong

We've tried to make the installation of the Developer's Kit distribution as painless as possible. Still, some complications may arise. Here are suggestions for dealing with some of them.

No customer ID for send-pr

Make sure the program `send-pr` knows your Cygnus customer identification code. You can install your customer ID by using the program `install-sid` as follows:

```
install-sid customer-ID
```

If you installed the Developer's Kit into a location other than the default, and you chose not to set up symbolic links pointing to the real installation location, you need to use the '--install-dir' option to `install-sid` as follows:

```
install-sid --install-dir=install-dir-prefix customer-ID
```

where *install-dir-prefix* points to the top level of the installation. Contact Cygnus Support at +1 415 903 1401 if you do not know your customer ID.

Not enough space

If you don't have enough space to install all of the tape distribution, you can instead extract only the compiled code, or only the source.

You can easily extract these components independently of one another by using the 'source' or 'bin' arguments to `Install`. See "Invoking the Install script," page 8.

No access to '/usr/cygnus'

If you can't sign on to an account with access to write in '`/usr`' or '`/usr/cygnus`', use the '`-installdir=`*directory*' option to `Install` to specify a different installation directory to which you *can* write. For example, if all the other installation defaults are right, you can execute something like '`./Install -tape=/dev/`*tape* `-installdir=`*mydir*'. You'll need to either override default paths for the pre-compiled tools, or else recompile the software. See "Running the programs," page 15, and "Links for easy access and updating," page 12, for details.

> *WARNING:* If you can't install in '`/usr/cygnus`' (or link your installation directory to that name), some of the defaults configured into the `progressive-94q4` distribution won't work. See "Running the programs," page 15, for information on overriding or reconfiguring these defaults.

Error messages from `Install`

The `Install` script checks for many errors and inconsistencies in the way its arguments are used. The messages are meant to be self-explanatory. Here is a list of a few messages where further information might be useful:

`Can not read from TAPE device,` *tape*

: The error message ends with the tape device `Install` was trying to use. Please check that it is the device you intended; possible causes of trouble might include leaving off the '`/dev/`' prefix at the front of the device name. A typographical error in the device name might also cause this problem.

  If the problem is neither of these, perhaps your tape device can't read our tape; see "Installing with a remote tape drive," page 14, for a discussion of how to use another machine's tape drive, or contact Cygnus Support.

`stdin: not in compressed format`

: You are probably not using the non–rewinding tape device. There are two files on each tape. The first is a `tar` file containing the `Install` script. The second is a compressed `tar` file containing everything else. Without using the non–rewinding device, there is no way to skip over the first file to begin reading the second.

`gcc: cannot exec cpp`

: If you've installed the binary distribution of the Developer's Kit software in a non-standard location, remember to set

your environment variable `GCC_EXEC_PREFIX` accordingly. See "Running the programs," page 15.

`. . . This is a problem.`
`Cannot cd to` *`installdir`*
`I do not know why I cannot create` *`installdir`*
`hello.c fails to run`
`test-ioctl.c fails to run`

These errors (the first covers anything that ends in '`This is a problem`') are from paranoia checks; they are issued for situations that other checks should have covered, or for unlikely situations that require further diagnosis.

If you get one of these messages, please call the Cygnus hotline, +1 415 903 1401, or send electronic mail to '`help@cygnus.com`'.

Developer's Kit installation on MS-DOS

This section describes the installation procedure for the Cygnus Support Developer's Kit distribution running on MS-DOS. For specific information about using this release with MS-DOS, see the MS-DOS specific developer's note, *Developing with DOS*.

MS-DOS installation directories

You may install the software in any directory. The `INSTALL` program assumes that you are installing in `C:\CYGNUS`; if you want to change this, you may do so at the beginning of the installation process.

If you are installing cross-development tools for more than one target, you *must* install them into different directories; otherwise, the second installation overwrites the first. We recommend you use a directory structure that has the target name built-in, such as:

`C:\CYGNUS\A29KUDI\. . .`	*AMD 29k cross-development toolkit*
`C:\CYGNUS\M68KCOFF\. . .`	*Motorola 68k cross-development toolkit*
`. . .`	

If you use this paradigm, remember to type

```
disk:\CYGNUS\target\SETENV
```

to reset your environment every time you switch targets.

Disk space

The total space required to extract and install binaries for all programs in the Developer's Kit is from 10 to 16 megabytes, depending on the target. The actual disk space required by the Developer's Kit is printed on the disk label. The `INSTALL` program dynamically compares the space available on your designated drive with the size of the installation before starting the installation. If you do not have enough space to install the binaries, the `INSTALL` program exits with an error message before writing anything.

Note that if you're using a disk compression utility like `STACKER` then the actual amount of disk space that you have available may be less than reported. This is because the compression utilities usually report the amount of free space available assuming that the data which would go into it would be compressed at least as well as the data already on the disk. This is important information if you're trying to install the tools onto a compressed disk with only just enough room for the installation. It could be that you run out of real disk space before the installation is complete because the compression utility couldn't do as good a job as it expected.

MS-DOS memory requirements

We do not recommend using the cross-development kit with less than four (4) megabytes of RAM.

We provide a MS-DOS extender with the cross-development kit for MS-DOS which swaps programs to disk when MS-DOS runs out of memory. To avoid excessive swapping you must have at least 2 megabytes of RAM to run G++ on a PC with MS-DOS.

If you've got more than 2 megabytes, the extra memory can be used as a disk cache to significantly improve performance.

Installation your Developer's Kit

We ship your Developer's Kit on a set of floppy disks. The `INSTALL` program is included on Disk 1.

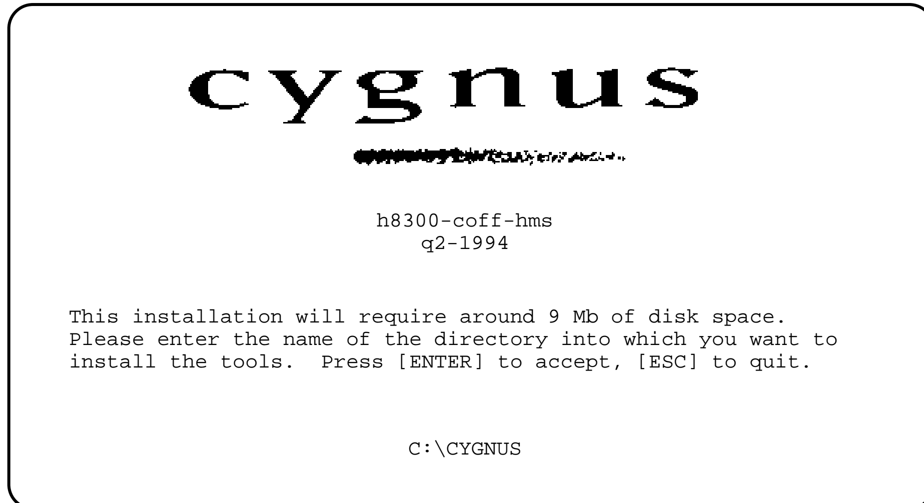
The files are stored on the floppies using Microsoft's `COMPRESS` program. If you prefer, you can install files without using the `INSTALL` program by just copying them into the right place on your hard drive and running `EXPAND` on each file. Since the files are stored on the floppy using their full name (not those marked as compressed by using Microsoft's '`.XX_`' naming convention) you *must* use a temporary file.

> *Warning!* If you have a program in your path called `EXPAND` and it's not the one provided by Microsoft, then you should either change your path to use only the Microsoft `EXPAND` program, or be certain that your `EXPAND` program can decompress files which have been compressed using Microsoft `COMPRESS`.

We show the system prompt as '`C:\>`' for the local hard disk drive, and '`A:\>`' for the local 3.5" floppy disk drive.

For these examples we assume that you install into a directory called '`C:\CYGNUS`' and that you use drive '`A:`' to read the installation floppies. Substitute other hard drives, installation directories, and floppy drives to match your environment.

The INSTALL program first prompts you for an installation directory:

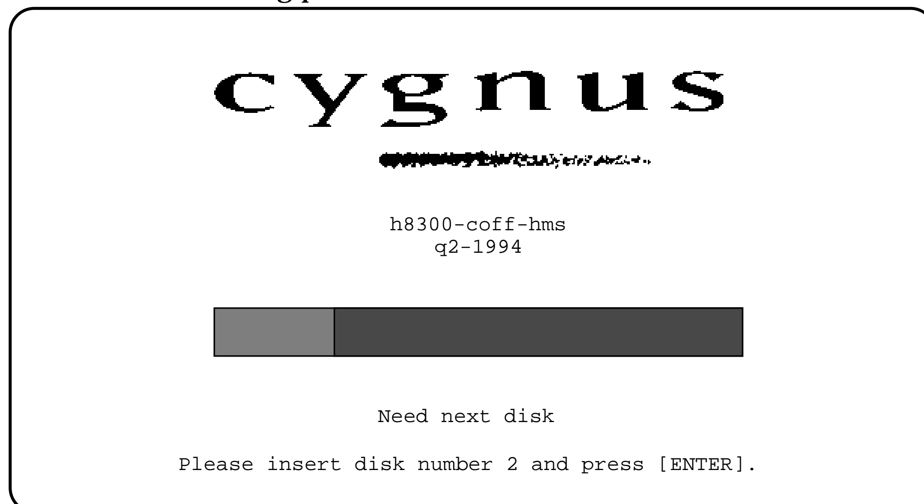


At this prompt, enter the name of the directory where you want the tools to be installed and press ENTER. INSTALL prompts you (assuming that you accept 'C:\CYGNUS'):

```
The installation will write into C:\CYGNUS.
Are you sure you want to continue [Y] or [N]
```

If you type N, INSTALL asks for another path name. If you type Y, INSTALL begins the installation.

The program draws a status bar, which fills up as INSTALL works. When the box is full, the installation is complete. INSTALL shows the name of the file being processed at the bottom of the screen.



INSTALL prompts you for each disk in order.

To use another installation directory, specify the path to your desired directory wherever the examples show 'C:\CYGNUS'

You must execute the batch file 'SETENV.BAT' before running the Developer's Kit. You can set the environment automatically whenever you boot the machine by putting the following line in your 'AUTOEXEC.BAT':

```
CALL C:\CYGNUS\SETENV.BAT
```

(If you install in a location other than 'C:\CYGNUS', be sure to specify the correct directory.)

Release contents

The programs in this Developer's Kit are shipped as binaries, preconfigured to run on Intel *x*86 PCs running standard MS-DOS.

The individual programs in the Developer's Kit are:

REL	Name of the release
MANIFEST	Disk contents manifest
README\COPYING	Information about copying this release
README\README	Last minute information
BIN\AR.EXE	Archive utility
BIN\AS.EXE	Assembler
BIN\ASYNCTSR.COM	Serial line driver TSR
BIN\CC1.EXE	C compiler
BIN\CC1PLUS.EXE	C++ compiler
BIN\CPP.EXE	C preprocessor
BIN\CXX.EXE	C++ compiler driver
BIN\CXXFILT.EXE	C++ symbol name filter
BIN\EMU387	387 emulator
BIN\GASP.EXE	Assembler preprocessor
BIN\GCC.EXE	C compiler driver
BIN\GDB.EXE	Debugger
BIN\GO32.EXE	dos extender
BIN\GXX.EXE	C++ compiler driver
BIN\INFO.EXE	Documentation browser
BIN\LD.EXE	Linker
BIN\MAKE.EXE	Recompilation director
BIN\NM.EXE	Symbol name utility
BIN\OBJCOPY.EXE	Object file copier and converter
BIN\OBJDUMP.EXE	Object file dumper
BIN\RANLIB.EXE	Archive indexer
BIN\SIZE.EXE	Object file size utility
BIN\STRINGS.EXE	Object file strings utility
BIN\STRIP.EXE	Object file symbol stripper
DEMO\HELLO.C	Demonstration program
DEMO\INIT.BAT	Demonstration initialzation batch file
DEMO\MAKEFILE	Makefile for demonstration
INSTALL.EXE	The install program
LIB\LIBC.A	ansi C library

LIB\LIBGCC.A	Compiler support library
LIB\LIBM.A	Maths library
INCLUDE_ANSI.H	Include files for C library
INFO*.INF	Online documentation, read with info.exe
LIB\LDSCRIPT\ELF32MIP.x	Linker information scripts

The included libraries and some utilities are different depending on which target this release is intended for. The file 'MANIFEST' in the root directory of the first installation disk contains a complete list of everything included in this release.

How to report bugs

If you find a bug in this release, please report it to Cygnus Support.

Use a copy of the Cygnus bug-report form to ensure that we can respond to your bug as quickly as possible. The file 'SENDPR.TXT' in the installation directory contains a blank copy of this form. To save time, customize this form ahead of time with your Cygnus customer ID, as described in the previous section.

The easiest way to report a bug is to fill in a copy of this form on your computer and send it via Internet electronic mail to 'bugs@cygnus.com'. Otherwise, you can print the file 'SENDPR.TXT', fill it in, and FAX the problem report to Cygnus at +1 415 903 0122 (Mountain View, California) or +1 617 629 3010 (Somerville, Massachusetts). Contact Cygnus if you have any trouble.

Cygnus Support
hotline: +1 415 903 1401
email: info@cygnus.com

Headquarters	*East Coast*
1937 Landings Drive	48 Grove St., Ste. 105
Mountain View, CA 94043 USA	Somerville, MA 02144 USA
+1 415 903 1400	+1 617 629 3000
+1 415 903 0122 fax	fax +1 617 629 3010

Source code for your Developer's Kit

The QIC-24 tape included with your Developer's Kit contains source code for all the programs. Most DOS systems cannot read this tape; you probably need to find a Unix system to read it.

You need about 71 megabytes of free disk space to hold the source code. To extract the source code into the current working directory on most Unix machines, execute a command like this:

```
dd if=tapedev | compress -d | tar xvf -
```

tapedev stands for the device name for the tape drive. For example, on most Sun workstations, the device name for the QIC-24 tape drive is '/dev/rst8'. Contact your system administrator for the correct tape device for your system.

Please contact Cygnus Support at +1 415 903 1401 if you would like the source code in another form.

Appendix A Platform names

Your tape is labeled to indicate the host (and target, if applicable) for which the binaries in the distribution are configured. The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, though the scheme is slightly different between hosts and targets.

Host names

The full naming scheme for hosts encodes three pieces of information in the following pattern:

architecture-vendor-os

For example, the full name for a Sun SPARCstation running SunOS 4.1.3 is

`sparc-sun-sunos4.1.3`

> *Warning:* `configure` can represent a very large number of combinations of architecture, vendor, and operating system. There is by no means support for all possible combinations!

The following combinations refer to hosts supported by Cygnus Support. Some common short aliases are included, but these may be obsolete in the future. (For a matrix which shows all supported host/target combinations, see section "Overview" in *Release Notes*.)

canonical name	*alias*	*platform*
sparc-sun-solaris2	sun4sol2	Sun 4 running Solaris 2
sparc-sun-sunos4.1.3	sun4	Sun-4 running SunOS 4
mips-dec-ultrix	decstation	DECstation
rs6000-ibm-aix	rs6000	IBM RS6000
mips-sgi-irix4	iris	SGI Iris running Irix 4
m68k-hp-hpux	hp300hpux	HP 9000/300
hppa1.1-hp-hpux	hp700	HP 9000/700
i386-unknown-sysv4		UnixWare
i386-lynx-lynxos	i386-lynx	Intel *x*86 Lynxos 2.2
m68k-lynx-lynxos	m68k-lynx	Motorola 68k Lynxos 2.2
sparc-lynx-lynxos	sparc-lynx	SPARC Lynxos 2.2
rs6000-lynx-lynxos	rs6000-lynx	IBM RS6000 Lynxos 2.2
alpha-dec-osf1.3		DEC Alpha running OSF/1 v1.3

Target names

If you have a cross-development tape, the label also indicates the target for that configuration. The pattern for target names is

architecture[-*vendor*]-*objfmt*

Target names differ slightly from host names in that the last variable indicates the object format rather than the operating system, and the second variable is often left out (this practice is becoming obsolete; in the future, all configuration names will be made up of three parts).

In cross-development configurations, each tool in the Developer's Kit is installed with the configured name of the target as a prefix. For example, if the C compiler is configured to generate COFF format code for the Motorola 680x0 family, the compiler is installed as '`m68k-coff-gcc`'.

> *Warning:* `configure` can represent a very large number of target name combinations of architecture, vendor, and object format. There is by no means support for all possible combinations!

This is a list of some of the more common targets supported by Cygnus Support. (Not all targets are supported on every host!) The list is not exhaustive; see section "Overview" in *Release Notes*, for an up-to-date matrix which shows the host/target combinations supported by Cygnus.

Motorola 68000 family

`m68k-aout`	`a.out` object code format
`m68k-coff`	COFF object code format
`m68k-vxworks`	VxWorks environment
`m68k-lynx`	Lynxos 2.2 environment

Motorola 88000 family

`m88k-coff`	COFF object code format

Intel 960 family

`i960-vxworks5.0`	VxWorks environment (`b.out` format)
`i960-vxworks5.1`	VxWorks environment (COFF format
`i960-intel-nindy`	Nindy monitor

AMD 29000 family

`a29k-amd-udi`	UDI monitor interface

To use the minimon *interface, use this configuration with the auxiliary program* MONTIP*, available from AMD.*

SPARC family

`sparc-vxworks`	VxWorks environment
`sparc-aout`	`a.out` object code format
`sparclite-aout`	`a.out` object code format
`sparclite-coff`	COFF object code format

Intel 80X86 family

`i386-aout`	`a.out` object code format
`i386-netware`	NetWare NLM
`i386-lynx`	LynxOS 2.2 environment

IDT/MIPS R3000

`mips-idt-ecoff`	IDT R3000, big endian ECOFF
`mipsel-idt-ecoff`	IDT R3000, little endian ECOFF
`mips64-idt-ecoff`	IDT R4000 ECOFF

Hitachi H8300

	`h8300-hms-coff`	COFF object code format

Hitachi SH

	`sh-hms-coff`	COFF object code format

Z8000

	`z8k-coff`	COFF object code format

config.guess

`config.guess` is a shell script which attempts to deduce the host type from which it is called, using system commands like `uname` if they are available. `config.guess` is remarkably adept at deciphering the proper configuration for your host; if you are building a tree to run on the same host on which you're building it, we recommend *not* specifying the *hosttype* argument.

`config.guess` is called by `configure`; you need never run it by hand, unless you're curious about the output.

Appendix B Cross-development environment

Using the Developer's Kit in one of the cross-development configurations usually requires some attention to setting up the target environment. (A cross-development configuration is used for developing software to run on a different machine (the *target*) from the development tools themselves (which run on the *host*)—for example, you might use a SPARCstation to generate and debug code for an AMD 29K-based board.)

To allow our tools to work with your target environment (except for real-time operating systems, which provide full operating system support), you need to set up:

- the C run-time environment (described below).
- *stubs*, or minimal versions of operating system subroutines for the C subroutine library. See section "System Calls" in *The Cygnus C Support Library*.
- a connection to the debugger. See section "Remote Debugging" in *Debugging with GDB*.

The C Run-Time Environment (`crt0`)

To link and run C or C++ programs, you need to define a small module (usually written in assembler as '`crt0.s`') that makes sure the hardware is initialized for C conventions before calling `main`.

There are some examples of '`crt0.s`' code (along with examples of system call stub code) available in the source code for your Developer's Kit. Look in the directories under:

```
installdir/progressive-94q4/src/newlib/libc/sys
```

(*installdir* refers to your installation directory, by default '`/usr/cygnus`'.) For example, look in '`.../sys/h8300hms`' for Hitachi H8/300 bare boards, or in '`.../sys/sparclite`' for the Fujitsu SPARClite board. More examples are available under the directory:

```
installdir/progressive-94q4/src/newlib/stub
```

To write your own '`crt0.s`', you need this information about your target:

- A memory map. What memory is available, and where?
- Which way does the stack grow?
- What output format do you use?

At a minimum, your '`crt0.s`' must do these things:

1. Define the symbol `start` ('`_start`' in assembler code). Execution begins at this symbol.

2. Set up the stack pointer '`sp`'. It is largely up to you to choose where to store your stack within the constraints of your target's memory map. Perhaps the simplest choice is to choose a fixed-size area somewhere in the uninitialized-data section (often called '`bss`'). Remember that whether you choose the low address or the high address in this area depends on the direction your stack grows.
3. Initialize all memory in the uninitialized-data ('`bss`') section to zero. The easiest way to do this is with the help of a linker script (see section "Command Language" in *Using LD*). Use a linker script to define symbols such as '`bss_start`' and '`bss_end`' to record the boundaries of this section; then you can use a '`for`' loop to initialize all memory between them in '`crt0.s`'.
4. Call `main`. Nothing else will!

A more complete '`crt0.s`' might also do the following:

5. Define an '`_exit`' subroutine (this is the C name; in your assembler code, use the label '`__exit`', with two leading underbars). Its precise behavior depends on the details of your system, and on your choice. Possibilities include trapping back to the boot monitor, if there is one; or to the loader, if there is no monitor; or even back to the symbol `start`.
6. If your target has no monitor to mediate communications with the debugger, you must set up the hardware exception handler in '`crt0.s`'. See section "The GDB remote serial protocol" in *Debugging with GDB*, for details on how to use the GDB generic remote-target facilities for this purpose.
7. Perform other hardware-dependent initialization; for example, initializing an MMU or an auxiliary floating-point chip.
8. Define low-level input and output subroutines. For example, '`crt0.s`' is a convenient place to define the minimal assembly-level routines described in section "System Calls" in *The Cygnus C Support Library*.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed

under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it

and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

GNU Online Documentation

Reading and Making Info files

For GNU Info version 2.9

Brian J. Fox (bfox@ai.mit.edu)

Edited by Roland H. Pesch (pesch@cygnus.com)
for Cygnus Support

Copyright © 1992, 1993, 1994, 1995 Free Software Foundation

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided 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 manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Table of Contents

GNU Online Documentation 1

1 Reading GNU Online Documentation 3

1.1 Command Line Options 3
1.2 Moving the Cursor.. 5
1.3 Moving Text Within a Window............................ 6
1.4 Selecting a New Node 7
1.5 Searching an Info File.................................. 9
1.6 Selecting Cross References 10
1.6.1 Parts of an Xref 10
1.6.2 Selecting Xrefs 11
1.7 Manipulating Multiple Windows 12
1.7.1 The Mode Line 12
1.7.2 Window Commands 13
1.7.3 The Echo Area 14
1.8 Printing Out Nodes 16
1.9 Miscellaneous Commands.............................. 17
1.10 Manipulating Variables 18

2 Making Info Files from Texinfo Files 23

2.1 Controlling Paragraph Formats......................... 23
2.2 Command Line Options for Makeinfo 23
2.3 What Makes a Valid Info File? 25
2.4 Defaulting the `Prev`, `Next`, and `Up` 26

Index 29

GNU Online Documentation

You can read the manuals for GNU software either on paper, as with any other manual, or as online *info files*, using an ordinary ASCII terminal.

You can browse through the online documentation with GNU Emacs, or with the program Info, a smaller program intended just for the purpose of viewing info files. This manual describes version 2.9 of Info.

Info files are generated by the program `makeinfo` from a *texinfo* source file. (Texinfo is a documentation markup language designed to allow the same source file to generate either printed or online documentation.) This manual describes version 1.51 of Makeinfo. The Texinfo language is described in a separate manual, *Texinfo: The GNU Documentation Format*.

1 Reading GNU Online Documentation

The info file version of a manual is organized into *nodes*, which usually correspond to the chapters and sections of the printed book. You can follow them in sequence, if you wish, just like in the printed book—but there are also other choices. Info files have menus that let you go quickly to the node that has the information you need. Info has “hot” references; if one section refers to another, you can tell Info to take you immediately to that other section—and you can get back again easily to take up your reading where you left off. Naturally, you can also search for particular words or phrases.

The best way to get started with the online documentation system is to use a programmed tutorial by running Info itself. You can get into Info by just typing its name—no options or arguments are necessary—at your shell’s prompt (shown as ‘`eg$`’ here):

```
eg$ info
```

Info displays its first screen, a menu of the documentation available, and awaits your input. Type the single letter

```
h
```

to request a tutorial, designed to teach you how to use Info.

If you already use Emacs, you may want to get into the documentation browsing mode, instead, by typing `C-h i` inside Emacs.

You can get out of Info at any time by typing the single letter `q`.

Info can also display a summary of all its commands at any time, when you type the single character `?`.

What is Info?

This text documents the use of the GNU Info program, version 2.9.

Info is a program which is used to view info files on an ASCII terminal. *info files* are the result of processing texinfo files with the program `makeinfo` or with the Emacs command `M-x texinfo-format-buffer`. Finally, *texinfo* is a documentation language which allows a printed manual and online documentation (an info file) to be produced from a single source file.

1.1 Command Line Options

GNU Info accepts several options to control the initial node being viewed, and to specify which directories to search for info files. Here is a template showing an invocation of GNU Info from the shell:

```
info [--option-name option-value] menu-item...
```

The following *option-names* are available when invoking Info from the shell:

`--directory` *directory-path*
`-d` *directory-path*
: Adds *directory-path* to the list of directory paths searched when Info needs to find a file. You may issue `--directory` multiple times; once for each directory which contains info files. Alternatively, you may specify a value for the environment variable `INFOPATH`; if `--directory` is not given, the value of `INFOPATH` is used. The value of `INFOPATH` is a colon separated list of directory names. If you do not supply `INFOPATH` or `--directory-path` a default path is used.

`--file` *filename*
`-f` *filename*
: Specifies a particular info file to visit. Instead of visiting the file `dir`, Info will start with `(`*filename*`)Top` as the first file and node.

`--node` *nodename*
`-n` *nodename*
: Specifies a particular node to visit in the initial file loaded. This is especially useful in conjunction with `--file`[1]. You may specify `--node` multiple times; for an interactive Info, each *nodename* is visited in its own window, for a non-interactive Info (such as when `--output` is given) each *nodename* is processed sequentially.

`--output` *filename*
`-o` *filename*
: Specify *filename* as the name of a file to output to. Each node that Info visits will be output to *filename* instead of interactively viewed. A value of `-` for *filename* specifies the standard output.

`--subnodes`
: This option only has meaning when given in conjunction with `--output`. It means to recursively output the nodes appearing in the menus of each node being output. Menu items which resolve to external info files are not output, and neither are menu items which are members of an index. Each node is only output once.

[1] Of course, you can specify both the file and node in a `--node` command; but don't forget to escape the open and close parentheses from the shell as in: `info --node '(emacs)Buffers'`

`--help`
`-h` Produces a relatively brief description of the available Info options.

`--version`
Prints the version information of Info and exits.

menu-item
Remaining arguments to Info are treated as the names of menu items. The first argument would be a menu item in the initial node visited, while the second argument would be a menu item in the first argument's node. You can easily move to the node of your choice by specifying the menu names which describe the path to that node. For example,

```
info emacs buffers
```

first selects the menu item '`Emacs`' in the node '`(dir)Top`', and then selects the menu item '`Buffers`' in the node '`(emacs)Top`'.

1.2 Moving the Cursor

Many people find that reading screens of text page by page is made easier when one is able to indicate particular pieces of text with some kind of pointing device. Since this is the case, GNU Info (both the Emacs and standalone versions) have several commands which allow you to move the cursor about the screen. The notation used in this manual to describe keystrokes is identical to the notation used within the Emacs manual, and the GNU Readline manual. See section "Character Conventions" in *the GNU Emacs Manual*, if you are unfamilar with the notation.

The following table lists the basic cursor movement commands in Info. Each entry consists of the key sequence you should type to execute the cursor movement, the `M-x`[2] command name (displayed in parentheses), and a short description of what the command does. All of the cursor motion commands can take an *numeric* argument (see Section 1.9 "Miscellaneous Commands," page 17), to find out how to supply them. With a numeric argument, the motion commands are simply executed that many times; for example, a numeric argument of 4 given to `next-line` causes the cursor to move down 4 lines. With a negative numeric argument, the motion is reversed; an argument of -4 given to the `next-line` command would cause the cursor to move *up* 4 lines.

[2] `M-x` is also a command; it invokes `execute-extended-command`. See section "Executing an extended command" in *the GNU Emacs Manual*, for more detailed information.

`C-n` (`next-line`)
: Moves the cursor down to the next line.

`C-p` (`prev-line`)
: Move the cursor up to the previous line.

`C-a` (`beginning-of-line`)
: Move the cursor to the start of the current line.

`C-e` (`end-of-line`)
: Moves the cursor to the end of the current line.

`C-f` (`forward-char`)
: Move the cursor forward a character.

`C-b` (`backward-char`)
: Move the cursor backward a character.

`M-f` (`forward-word`)
: Moves the cursor forward a word.

`M-b` (`backward-word`)
: Moves the cursor backward a word.

`M-<` (`beginning-of-node`)
`b`
: Moves the cursor to the start of the current node.

`M->` (`end-of-node`)
: Moves the cursor to the end of the current node.

`M-r` (`move-to-window-line`)
: Moves the cursor to a specific line of the window. Without a numeric argument, `M-r` moves the cursor to the start of the line in the center of the window. With a numeric argument of *n*, `M-r` moves the cursor to the start of the *n*th line in the window.

1.3 Moving Text Within a Window

Sometimes you are looking at a screenful of text, and only part of the current paragraph you are reading is visible on the screen. The commands detailed in this section are used to shift which part of the current node is visible on the screen.

`SPC` (`scroll-forward`)
`C-v`
: Shift the text in this window up. That is, show more of the node which is currently below the bottom of the window. With a numeric argument, show that many more lines at the bottom of the window; a numeric argument of 4 would shift all of the text in the window up 4 lines (discarding the top

4 lines), and show you four new lines at the bottom of the window. Without a numeric argument, `SPC` takes the bottom two lines of the window and places them at the top of the window, redisplaying almost a completely new screenful of lines.

`M-v`

`DEL` (`scroll-backward`)
: Shift the text in this window down. The inverse of `scroll-forward`.

The `scroll-forward` and `scroll-backward` commands can also move forward and backward through the node structure of the file. If you press `SPC` while viewing the end of a node, or `DEL` while viewing the beginning of a node, what happens is controlled by the variable `scroll-behaviour`. See Section 1.10 “Manipulating Variables,” page 18, for more information.

`C-l` (`redraw-display`)
: Redraw the display from scratch, or shift the line containing the cursor to a specified location. With no numeric argument, ‘`C-l`’ clears the screen, and then redraws its entire contents. Given a numeric argument of *n*, the line containing the cursor is shifted so that it is on the *n*th line of the window.

`C-x w` (`toggle-wrap`)
: Toggles the state of line wrapping in the current window. Normally, lines which are longer than the screen width *wrap*, i.e., they are continued on the next line. Lines which wrap have a ‘\’ appearing in the rightmost column of the screen. You can cause such lines to be terminated at the rightmost column by changing the state of line wrapping in the window with `C-x w`. When a line which needs more space than one screen width to display is displayed, a ‘`$`’ appears in the rightmost column of the screen, and the remainder of the line is invisible.

1.4 Selecting a New Node

This section details the numerous Info commands which select a new node to view in the current window.

The most basic node commands are ‘`n`’, ‘`p`’, ‘`u`’, and ‘`l`’.

When you are viewing a node, the top line of the node contains some Info *pointers* which describe where the next, previous, and up nodes are. Info uses this line to move about the node structure of the file when you use the following commands:

`n` (`next-node`)
    Selects the 'Next' node.

`p` (`prev-node`)
    Selects the 'Prev' node.

`u` (`up-node`)
    Selects the 'Up' node.

You can easily select a node that you have already viewed in this window by using the '`l`' command – this name stands for "last", and actually moves through the list of already visited nodes for this window. '`l`' with a negative numeric argument moves forward through the history of nodes for this window, so you can quickly step between two adjacent (in viewing history) nodes.

`l` (`history-node`)
    Selects the most recently selected node in this window.

Two additional commands make it easy to select the most commonly selected nodes; they are '`t`' and '`d`'.

`t` (`top-node`)
    Selects the node '`Top`' in the current info file.

`d` (`dir-node`)
    Selects the directory node (i.e., the node '`(dir)`').

Here are some other commands which immediately result in the selection of a different node in the current window:

`<` (`first-node`)
    Selects the first node which appears in this file. This node is most often '`Top`', but it doesn't have to be.

`>` (`last-node`)
    Selects the last node which appears in this file.

`]` (`global-next-node`)
    Moves forward or down through node structure. If the node that you are currently viewing has a '`Next`' pointer, that node is selected. Otherwise, if this node has a menu, the first menu item is selected. If there is no '`Next`' and no menu, the same process is tried with the '`Up`' node of this node.

`[` (`global-prev-node`)
    Moves backward or up through node structure. If the node that you are currently viewing has a '`Prev`' pointer, that node is selected. Otherwise, if the node has an '`Up`' pointer, that node is selected, and if it has a menu, the last item in the menu is selected.

You can get the same behaviour as `global-next-node` and `global-prev-node` while simply scrolling through the file with `SPC` and `DEL`; See Section 1.10 "`scroll-behaviour`," page 18, for more information.

`g` (`goto-node`)
: Reads the name of a node and selects it. No completion is done while reading the node name, since the desired node may reside in a separate file. The node must be typed exactly as it appears in the info file. A file name may be included as with any node specification, for example

```
g(emacs)Buffers
```

finds the node '`Buffers`' in the info file '`emacs`'.

`C-x k` (`kill-node`)
: Kills a node. The node name is prompted for in the echo area, with a default of the current node. *Killing* a node means that Info tries hard to forget about it, removing it from the list of history nodes kept for the window where that node is found. Another node is selected in the window which contained the killed node.

`C-x C-f` (`view-file`)
: Reads the name of a file and selects the entire file. The command

```
C-x C-f filename
```

is equivalent to typing

```
g(filename)*
```

`C-x C-b` (`list-visited-nodes`)
: Makes a window containing a menu of all of the currently visited nodes. This window becomes the selected window, and you may use the standard Info commands within it.

`C-x b` (`select-visited-node`)
: Selects a node which has been previously visited in a visible window. This is similar to '`C-x C-b`' followed by '`m`', but no window is created.

1.5 Searching an Info File

GNU Info allows you to search for a sequence of characters throughout an entire info file, search through the indices of an info file, or find areas within an info file which discuss a particular topic.

`s` (`search`)
: Reads a string in the echo area and searches for it.

`C-s` (`isearch-forward`)
: Interactively searches forward through the info file for a string as you type it.

`C-r` (`isearch-backward`)
: Interactively searches backward through the info file for a string as you type it.

`i` (`index-search`)
: Looks up a string in the indices for this info file, and selects a node where the found index entry points to.

`,` (`next-index-match`)
: Moves to the node containing the next matching index item from the last 'i' command.

The most basic searching command is 's' (`search`). The 's' command prompts you for a string in the echo area, and then searches the remainder of the info file for an ocurrence of that string. If the string is found, the node containing it is selected, and the cursor is left positioned at the start of the found string. Subsequent 's' commands show you the default search string within '[' and ']'; pressing RET instead of typing a new string will use the default search string.

Incremental searching is similar to basic searching, but the string is looked up while you are typing it, instead of waiting until the entire search string has been specified.

1.6 Selecting Cross References

We have already discussed the 'Next', 'Prev', and 'Up' pointers which appear at the top of a node. In addition to these pointers, a node may contain other pointers which refer you to a different node, perhaps in another info file. Such pointers are called *cross references*, or *xrefs* for short.

1.6.1 Parts of an Xref

Cross references have two major parts: the first part is called the *label*; it is the name that you can use to refer to the cross reference, and the second is the *target*; it is the full name of the node that the cross reference points to.

The target is separated from the label by a colon ':'; first the label appears, and then the target. For example, in the sample menu cross reference below, the single colon separates the label from the target.

```
* Foo Label: Foo Target.  More information about Foo.
```

Note the '.' which ends the name of the target. The '.' is not part of the target; it serves only to let Info know where the target name ends.

A shorthand way of specifying references allows two adjacent colons to stand for a target name which is the same as the label name:

```
* Foo Commands::  Commands pertaining to Foo.
```

In the above example, the name of the target is the same as the name of the label, in this case `Foo Commands`.

You will normally see two types of cross references while viewing nodes: *menu* references, and *note* references. Menu references appear within a node's menu; they begin with a '`*`' at the beginning of a line, and continue with a label, a target, and a comment which describes what the contents of the node pointed to contains.

Note references appear within the body of the node text; they begin with `*Note`, and continue with a label and a target.

Like '`Next`', '`Prev`' and '`Up`' pointers, cross references can point to any valid node. They are used to refer you to a place where more detailed information can be found on a particular subject. Here is a cross reference which points to a node within the Texinfo documentation: See section "Writing an Xref" in *the Texinfo Manual*, for more information on creating your own texinfo cross references.

1.6.2 Selecting Xrefs

The following table lists the Info commands which operate on menu items.

`1` (`menu-digit`)
`2 ... 9`
: Within an Info window, pressing a single digit, (such as '`1`'), selects that menu item, and places its node in the current window. For convenience, there is one exception; pressing '`0`' selects the *last* item in the node's menu.

`0` (`last-menu-item`)
: Select the last item in the current node's menu.

`m` (`menu-item`)
: Reads the name of a menu item in the echo area and selects its node. Completion is available while reading the menu label.

`M-x find-menu`
: Moves the cursor to the start of this node's menu.

This table lists the Info commands which operate on note cross references.

`f` (`xref-item`)
`r`
: Reads the name of a note cross reference in the echo area and selects its node. Completion is available while reading the cross reference label.

Finally, the next few commands operate on menu or note references alike:

`TAB` (`move-to-next-xref`)
: Moves the cursor to the start of the next nearest menu item or note reference in this node. You can then use `RET` (`select-reference-this-line` to select the menu or note reference.

`M-TAB` (`move-to-prev-xref`)
: Moves the cursor the start of the nearest previous menu item or note reference in this node.

`RET` (`select-reference-this-line`)
: Selects the menu item or note reference appearing on this line.

1.7 Manipulating Multiple Windows

A *window* is a place to show the text of a node. Windows have a view area where the text of the node is displayed, and an associated *mode line*, which briefly describes the node being viewed.

GNU Info supports multiple windows appearing in a single screen; each window is separated from the next by its modeline. At any time, there is only one *active* window, that is, the window in which the cursor appears. There are commands available for creating windows, changing the size of windows, selecting which window is active, and for deleting windows.

1.7.1 The Mode Line

A *mode line* is a line of inverse video which appears at the bottom of an info window. It describes the contents of the window just above it; this information includes the name of the file and node appearing in that window, the number of screen lines it takes to display the node, and the percentage of text that is above the top of the window. It can also tell you if the indirect tags table for this info file needs to be updated, and whether or not the info file was compressed when stored on disk.

Here is a sample mode line for a window containing an uncompressed file named '`dir`', showing the node '`Top`'.

```
-----Info: (dir)Top, 40 lines --Top-------------------------
            ^^   ^   ^^^        ^^
          (file)Node #lines    where
```

When a node comes from a file which is compressed on disk, this is indicated in the mode line with two small 'z''s. In addition, if the info file containing the node has been split into subfiles, the name of the subfile containing the node appears in the modeline as well:

```
--zz-Info: (emacs)Top, 291 lines --Top-- Subfile: emacs-1.Z-
```

When Info makes a node internally, such that there is no corresponding info file on disk, the name of the node is surrounded by asterisks ('*'). The name itself tells you what the contents of the window are; the sample mode line below shows an internally constructed node showing possible completions:

```
-----Info: *Completions*, 7 lines --All---------------------
```

1.7.2 Window Commands

It can be convenient to view more than one node at a time. To allow this, Info can display more than one *window*. Each window has its own mode line (see Section 1.7.1 "The Mode Line," page 12) and history of nodes viewed in that window (see Section 1.4 "history-node," page 7).

C-x o (next-window)
: Selects the next window on the screen. Note that the echo area can only be selected if it is already in use, and you have left it temporarily. Normally, 'C-x o' simply moves the cursor into the next window on the screen, or if you are already within the last window, into the first window on the screen. Given a numeric argument, 'C-x o' moves over that many windows. A negative argument causes 'C-x o' to select the previous window on the screen.

M-x prev-window
: Selects the previous window on the screen. This is identical to 'C-x o' with a negative argument.

C-x 2 (split-window)
: Splits the current window into two windows, both showing the same node. Each window is one half the size of the original window, and the cursor remains in the original window. The variable automatic-tiling can cause all of the windows on the screen to be resized for you automatically, please see Section 1.10 "automatic-tiling," page 18 for more information.

`C-x 0` (`delete-window`)
Deletes the current window from the screen. If you have made too many windows and your screen appears cluttered, this is the way to get rid of some of them.

`C-x 1` (`keep-one-window`)
Deletes all of the windows excepting the current one.

`ESC C-v` (`scroll-other-window`)
Scrolls the other window, in the same fashion that '`C-v`' might scroll the current window. Given a negative argument, the "other" window is scrolled backward.

`C-x ^` (`grow-window`)
Grows (or shrinks) the current window. Given a numeric argument, grows the current window that many lines; with a negative numeric argument, the window is shrunk instead.

`C-x t` (`tile-windows`)
Divides the available screen space among all of the visible windows. Each window is given an equal portion of the screen in which to display its contents. The variable `automatic-tiling` can cause `tile-windows` to be called when a window is created or deleted. See Section 1.10 "`automatic-tiling`," page 18.

1.7.3 The Echo Area

The *echo area* is a one line window which appears at the bottom of the screen. It is used to display informative or error messages, and to read lines of input from you when that is necessary. Almost all of the commands available in the echo area are identical to their Emacs counterparts, so please refer to that documentation for greater depth of discussion on the concepts of editing a line of text. The following table briefly lists the commands that are available while input is being read in the echo area:

`C-f` (`echo-area-forward`)
Moves forward a character.

`C-b` (`echo-area-backward`)
Moves backward a character.

`C-a` (`echo-area-beg-of-line`)
Moves to the start of the input line.

`C-e` (`echo-area-end-of-line`)
Moves to the end of the input line.

`M-f` (`echo-area-forward-word`)
  Moves forward a word.

`M-b` (`echo-area-backward-word`)
  Moves backward a word.

`C-d` (`echo-area-delete`)
  Deletes the character under the cursor.

`DEL` (`echo-area-rubout`)
  Deletes the character behind the cursor.

`C-g` (`echo-area-abort`)
  Cancels or quits the current operation. If completion is being read, '`C-g`' discards the text of the input line which does not match any completion. If the input line is empty, '`C-g`' aborts the calling function.

`RET` (`echo-area-newline`)
  Accepts (or forces completion of) the current input line.

`C-q` (`echo-area-quoted-insert`)
  Inserts the next character verbatim. This is how you can insert control characters into a search string, for example.

`printing character` (`echo-area-insert`)
  Inserts the character.

`M-TAB` (`echo-area-tab-insert`)
  Inserts a TAB character.

`C-t` (`echo-area-transpose-chars`)
  Transposes the characters at the cursor.

The next group of commands deal with *killing*, and *yanking* text. For an in depth discussion of killing and yanking, see section "Killing and Deleting" in *the GNU Emacs Manual*

`M-d` (`echo-area-kill-word`)
  Kills the word following the cursor.

`M-DEL` (`echo-area-backward-kill-word`)
  Kills the word preceding the cursor.

`C-k` (`echo-area-kill-line`)
  Kills the text from the cursor to the end of the line.

`C-x DEL` (`echo-area-backward-kill-line`)
  Kills the text from the cursor to the beginning of the line.

`C-y` (`echo-area-yank`)
  Yanks back the contents of the last kill.

`M-y` (`echo-area-yank-pop`)
: Yanks back a previous kill, removing the last yanked text first.

Sometimes when reading input in the echo area, the command that needed input will only accept one of a list of several choices. The choices represent the *possible completions*, and you must respond with one of them. Since there are a limited number of responses you can make, Info allows you to abbreviate what you type, only typing as much of the response as is necessary to uniquely identify it. In addition, you can request Info to fill in as much of the response as is possible; this is called *completion*.

The following commands are available when completing in the echo area:

`TAB` (`echo-area-complete`)
`SPC`
: Inserts as much of a completion as is possible.

`?` (`echo-area-possible-completions`)
: Displays a window containing a list of the possible completions of what you have typed so far. For example, if the available choices are:

```
bar
foliate
food
forget
```

and you have typed an '`f`', followed by '`?`', the possible completions would contain:

```
foliate
food
forget
```

i.e., all of the choices which begin with '`f`'. Pressing `SPC` or `TAB` would result in '`fo`' appearing in the echo area, since all of the choices which begin with '`f`' continue with '`o`'. Now, typing '`l`' followed by '`TAB`' results in '`foliate`' appearing in the echo area, since that is the only choice which begins with '`fol`'.

`ESC C-v` (`echo-area-scroll-completions-window`)
: Scrolls the completions window, if that is visible, or the "other" window if not.

1.8 Printing Out Nodes

You may wish to print out the contents of a node as a quick reference document for later use. Info provides you with a command for doing this.

In general, we recommend that you use TEX to format the document and print sections of it, by running `tex` on the texinfo source file.

`M-x print-node`
: Pipes the contents of the current node through the command in the environment variable `INFO_PRINT_COMMAND`. If the variable doesn't exist, the node is simply piped to `lpr`.

1.9 Miscellaneous Commands

GNU Info contains several commands which self-document GNU Info:

`M-x describe-command`
: Reads the name of an Info command in the echo area and then displays a brief description of what that command does.

`M-x describe-key`
: Reads a key sequence in the echo area, and then displays the name and documentation of the Info command that the key sequence invokes.

`M-x describe-variable`
: Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

`M-x where-is`
: Reads the name of an Info command in the echo area, and then displays a key sequence which can be typed in order to invoke that command.

`C-h` (`get-help-window`)
`?`
: Creates (or moves into) the window displaying `*Help*`, and places a node containing a quick reference card into it. This window displays the most concise information about GNU Info available.

`h` (`get-info-help-node`)
: Tries hard to visit the node `(info)Help`. The info file '`info.texi`' distributed with GNU Info contains this node. Of course, the file must first be processed with `makeinfo`, and then placed into the location of your info directory.

Here are the commands for creating a numeric argument:

`C-u` (`universal-argument`)
: Starts (or multiplies by 4) the current numeric argument. '`C-u`' is a good way to give a small numeric argument to cursor movement or scrolling commands; '`C-u C-v`' scrolls the screen 4 lines, while '`C-u C-u C-n`' moves the cursor down 16 lines.

`M-1` (`add-digit-to-numeric-arg`)
`M-2 ... M-9`
Adds the digit value of the invoking key to the current numeric argument. Once Info is reading a numeric argument, you may just type the digits of the argument, without the Meta prefix. For example, you might give '`C-l`' a numeric argument of 32 by typing:

```
C-u 3 2 C-l
```

or

```
M-3 2 C-l
```

'`C-g`' is used to abort the reading of a multi-character key sequence, to cancel lengthy operations (such as multi-file searches) and to cancel reading input in the echo area.

`C-g` (`abort-key`)
Cancels current operation.

The '`q`' command of Info simply quits running Info.

`q` (`quit`) Exits GNU Info.

If the operating system tells GNU Info that the screen is 60 lines tall, and it is actually only 40 lines tall, here is a way to tell Info that the operating system is correct.

`M-x set-screen-height`
Reads a height value in the echo area and sets the height of the displayed screen to that value.

Finally, Info provides a convenient way to display footnotes which might be associated with the current node that you are viewing:

`ESC C-f` (`show-footnotes`)
Shows the footnotes (if any) associated with the current node in another window. You can have Info automatically display the footnotes associated with a node when the node is selected by setting the variable `automatic-footnotes`. See Section 1.10 "`automatic-footnotes`," page 18.

1.10 Manipulating Variables

GNU Info contains several *variables* whose values are looked at by various Info commands. You can change the values of these variables, and thus change the behaviour of Info to more closely match your environment and info file reading manner.

`M-x set-variable`
: Reads the name of a variable, and the value for it, in the echo area and then sets the variable to that value. Completion is available when reading the variable name; often, completion is available when reading the value to give to the variable, but that depends on the variable itself. If a variable does *not* supply multiple choices to complete over, it expects a numeric value.

`M-x describe-variable`
: Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

Here is a list of the variables that you can set in Info.

`automatic-footnotes`
: When set to `On`, footnotes appear and disappear automatically. This variable is `On` by default. When a node is selected, a window containing the footnotes which appear in that node is created, and the footnotes are displayed within the new window. The window that Info creates to contain the footnotes is called '`*Footnotes*`'. If a node is selected which contains no footnotes, and a '`*Footnotes*`' window is on the screen, the '`*Footnotes*`' window is deleted. Footnote windows created in this fashion are not automatically tiled so that they can use as little of the display as is possible.

`automatic-tiling`
: When set to `On`, creating or deleting a window resizes other windows. This variable is `Off` by default. Normally, typing '`C-x 2`' divides the current window into two equal parts. When `automatic-tiling` is set to `On`, all of the windows are resized automatically, keeping an equal number of lines visible in each window. There are exceptions to the automatic tiling; specifically, the windows '`*Completions*`' and '`*Footnotes*`' are *not* resized through automatic tiling; they remain their original size.

`visible-bell`
: When set to `On`, GNU Info attempts to flash the screen instead of ringing the bell. This variable is `Off` by default. Of course, Info can only flash the screen if the terminal allows it; in the case that the terminal does not allow it, the setting of this variable has no effect. However, you can make Info perform quietly by setting the `errors-ring-bell` variable to `Off`.

errors-ring-bell
: When set to On, errors cause the bell to ring. The default setting of this variable is On.

gc-compressed-files
: When set to On, Info garbage collects files which had to be uncompressed. The default value of this variable is Off. Whenever a node is visited in Info, the info file containing that node is read into core, and Info reads information about the tags and nodes contained in that file. Once the tags information is read by Info, it is never forgotten. However, the actual text of the nodes does not need to remain in core unless a particular info window needs it. For non-compressed files, the text of the nodes does not remain in core when it is no longer in use. But de-compressing a file can be a time consuming operation, and so Info tries hard not to do it twice. gc-compressed-files tells Info it is okay to garbage collect the text of the nodes of a file which was compressed on disk.

show-index-match
: When set to On, the portion of the matched search string is highlighted in the message which explains where the matched search string was found. The default value of this variable is On. When Info displays the location where an index match was found, (see Section 1.5 "next-index-match," page 9), the portion of the string that you had typed is highlighted by displaying it in the inverse case from its surrounding characters.

scroll-behaviour
: Controls what happens when forward scrolling is requested at the end of a node, or when backward scrolling is requested at the beginning of a node. The default value for this variable is Continuous. There are three possible values for this variable:

    Continuous
    : Tries to get the first item in this node's menu, or failing that, the 'Next' node, or failing that, the 'Next' of the 'Up'. This behaviour is identical to using the ']' (global-next-node) and '[' (global-prev-node) commands.

    Next Only
    : Only tries to get the 'Next' node.

`Page Only`
: Simply gives up, changing nothing. If `scroll-behaviour` is `Page Only`, no scrolling command can change the node that is being viewed.

`scroll-step`
: The number of lines to scroll when the cursor moves out of the window. Scrolling happens automatically if the cursor has moved out of the visible portion of the node text when it is time to display. Usually the scrolling is done so as to put the cursor on the center line of the current window. However, if the variable `scroll-step` has a nonzero value, Info attempts to scroll the node text by that many lines; if that is enough to bring the cursor back into the window, that is what is done. The default value of this variable is 0, thus placing the cursor (and the text it is attached to) in the center of the window. Setting this variable to 1 causes a kind of "smooth scrolling" which some people prefer.

`ISO-Latin`
: When set to `On`, Info accepts and displays ISO Latin characters. By default, Info assumes an ASCII character set. `ISO-Latin` tells Info that it is running in an environment where the European standard character set is in use, and allows you to input such characters to Info, as well as display them.

2 Making Info Files from Texinfo Files

Makeinfo is the program that builds info files from texinfo files. Be,fore reading this chapter, you should be familiar with reading info files.

If you want to run Makeinfo on a texinfo file prepared by someone else, this chapter contains all you need to know.

However, to write your own texinfo files, you should also read the Texinfo manual. See section "Texinfo" in *Texinfo—the GNU Documentation Format*.

2.1 Controlling Paragraph Formats

In general, Makeinfo *fills* the paragraphs that it outputs to the info file. Filling is the process of breaking up and connecting lines such that the output is nearly justified. With Makeinfo, you can control:

- The width of each paragraph (the *fill-column*).
- The amount of indentation that the first line of the paragraph receives (the *paragraph-indentation*).

2.2 Command Line Options for Makeinfo

The following command line options are available for Makeinfo.

`-I` *dir*
: Adds *dir* to the directory search list for finding files which are included with the `@include` command. By default, only the current directory is searched.

`-D` *var*
: Defines the texinfo flag *var*. This is equivalent to '`@set` *var*' in the texinfo file.

`-U` *var*
: Makes the texinfo flag *var* undefined. This is equivalent to '`@clear` *var*' in the texinfo file.

`--error-limit` *num*
: Sets the maximum number of errors that Makeinfo will print before exiting (on the assumption that continuing would be useless). The default number of errors printed before Makeinfo gives up on processing the input file is 100.

`--fill-column` *num*
: Specifies the maximum right-hand edge of a line. Paragraphs that are filled will be filled to this width. The default value for `fill-column` is 72.

`--footnote-style style`
: Sets the footnote style to *style*. *style* should either be '`separate`' to have Makeinfo create a separate node containing the footnotes which appear in the current node, or '`end`' to have Makeinfo place the footnotes at the end of the current node.

`--no-headers`
: Suppress the generation of menus and node headers. This option is useful together with the '`--output file`' and '`--no-split`' options (see below) to produce a simple formatted file (suitable for printing on a dumb printer) from texinfo source. If you do not have TEX, these two options may allow you to get readable hard copy.

`--no-split`
: Suppress the splitting stage of Makeinfo. In general, large output files (where the size is greater than 70k bytes) are split into smaller subfiles, each one apporximately 50k bytes. If you specify '`--no-split`', Makeinfo will not split up the output file.

`--no-pointer-validate`
`--no-validate`
: Suppress the validation phase of Makeinfo. Normally, after the file is processed, some consistency checks are made to ensure that cross references can be resolved, etc. See Section 2.3 "What Makes a Valid Info File?," page 25.

`--no-warn`
: Suppress the output of warning messages. This does *not* suppress the output of error messages, simply warnings. You might want this if the file you are creating has texinfo examples in it, and the nodes that are referenced don't actually exist.

`--no-number-footnotes`
: Suppress the automatic numbering of footnotes. The default is to number each footnote sequentially in a single node, resetting the current footnote number to 1 at the start of each node.

`--output file`
`-o file`
: Specify that the output should be directed to *file* instead of the file name specified in the `@setfilename` command found in the texinfo source. *file* can be the special token '`-`', which specifies standard output.

`--paragraph-indent` *num*
: Sets the paragraph indentation to *num*. The value of *num* is interpreted as follows:

  - A value of 0 (or '`none`') means not to change the existing indentation (in the source file) at the start of paragraphs.
  - A value less than zero means to indent paragraph starts to column zero by deleting any existing indentation.
  - A value greater than zero is the number of spaces to leave at the front of each paragraph start.

`--reference-limit` *num*
: When a node has many references in a single texinfo file, this may indicate an error in the structure of the file. *num* is the number of times a given node may be referenced (with `@prev`, `@next`, `@note`, or appearing in an `@menu`, for example) before Makeinfo prints a warning message about it.

`--verbose`
: Causes Makeinfo to inform you as to what it is doing. Normally Makeinfo only outputs text if there are errors or warnings.

`--version`
: Displays the Makeinfo version number.

2.3 What Makes a Valid Info File?

If you have not used '`--no-pointer-validate`' to suppress validation, Makeinfo will check the validity of the final info file. Mostly, this means ensuring that nodes you have referenced really exist. Here is a complete list of what is checked:

1. If a node reference such as `Prev`, `Next` or `Up` is a reference to a node in this file (i.e., not an external reference such as '`(DIR)`'), then the referenced node must exist.
2. In a given node, if the node referenced by the `Prev` is different than the node referenced by the `Up`, then the node referenced by the `Prev` must have a `Next` which references this node.
3. Every node except `Top` must have an `Up` field.
4. The node referenced by `Up` must contain a reference to this node, other than a `Next` reference. Obviously, this includes menu items and followed references.
5. If the `Next` reference is not the same as the `Next` reference of the `Up` reference, then the node referenced by `Next` must have a `Prev`

reference pointing back at this node. This rule still allows the last node in a section to point to the first node of the next chapter.

2.4 Defaulting the `Prev`, `Next`, and `Up`

If you write the `@node` commands in your texinfo source file without `Next`, `Prev`, and `Up` pointers, Makeinfo will fill in the pointers from context (by reference to the menus in your source file).

Although the definition of an info file allows a great deal of flexibility, there are some conventions that you are urged to follow. By letting Makeinfo default the `Next`, `Prev`, and `Up` pointers you can follow these conventions with a minimum of effort.

A common error ocurrs when adding a new node to a menu; often the nodes which are referenced in the menu do not point to each other in the same order as they appear in the menu.

Makeinfo node defaulting helps with this particular problem by not requiring any explicit information beyond adding the new node (so long as you do include it in a menu).

The node to receive the defaulted pointers must be followed immediately by a sectioning command, such as `@chapter` or `@section`, and must appear in a menu that is one sectioning level or more above the sectioning level that this node is to have.

Here is an example of how to use this feature.

```
@setfilename default-nodes.info
@node Top
@chapter Introduction
@menu
* foo::  the foo node
* bar::  the bar node
@end menu

@node foo
@section foo
this is the foo node.

@node bar
@section Bar
This is the Bar node.
@bye
```

produces

```
Info file default-nodes.info, produced by Makeinfo, -*- Text -*-

from input file default-nodes.texinfo.

File: default-nodes.info,  Node: Top
```

```
Introduction
************

* Menu:

* foo::  the foo node
* bar::  the bar node

File: default-nodes.info,  Node: foo,  Next: bar,  Up: Top

foo
===

this is the foo node.

File: default-nodes.info,  Node: bar,  Prev: foo,  Up: Top

Bar
===

This is the Bar node.
```

Index

,

, ... 10

-

--subnodes, command line option 4

?

?, in Info windows 17
?, in the echo area 16

[

[.......... 8

]

] 8

>

> 8

<

< 8

0

0, in Info windows 11

1

1 ... 9, in Info windows 11

A

abort-key 18
add-digit-to-numeric-arg 18
arguments, command line 3
automatic-footnotes 19
automatic-tiling 19

B

b, in Info winows 6
backward-char 6
backward-word 6
beginning-of-line 6
beginning-of-node 6

C

C-a, in Info windows 6
C-a, in the echo area 14
C-b, in Info windows 6
C-b, in the echo area 14
C-d, in the echo area 15
C-e, in Info windows 6
C-e, in the echo area 14
C-f, in Info windows 6
C-f, in the echo area 14
C-g, in Info windows 18
C-g, in the echo area 15
C-h 17
C-k, in the echo area 15
C-l 7
C-n 6
C-p 6
C-q, in the echo area 15
C-r 10
C-s 10
C-t, in the echo area 15
C-u 17
C-v 6
C-w 7
C-x ^ 14
C-x 0 14
C-x 1 14
C-x 2 13
C-x b 9
C-x C-b 9
C-x C-f 9
C-x DEL, in the echo area 15
C-x k 9
C-x o 13
C-x t 14
C-y, in the echo area 15
cancelling the current operation 18
cancelling typeahead 18
command line options 3
command summary online, Info 3
commands, describing 17

cursor, moving 5

D

d 8
DEL, in Info windows 7
DEL, in the echo area 15
delete-window 14
describe-command 17
describe-key 17
describe-variable 19
dir-node 8
directory path 4

E

echo area 14
echo-area-abort 15
echo-area-backward 14
echo-area-backward-kill-line 15
echo-area-backward-kill-word 15
echo-area-backward-word 15
echo-area-beg-of-line 14
echo-area-complete 16
echo-area-delete 15
echo-area-end-of-line 14
echo-area-forward 14
echo-area-forward-word 15
echo-area-insert 15
echo-area-kill-line 15
echo-area-kill-word 15
echo-area-newline 15
echo-area-possible-completions 16
echo-area-quoted-insert 15
echo-area-rubout 15
echo-area-scroll-completions-window 16
echo-area-tab-insert 15
echo-area-transpose-chars 15
echo-area-yank 15
echo-area-yank-pop 16
end-of-line 6
end-of-node 6
errors-ring-bell 20
ESC C-f 18
ESC C-v, in Info windows 14
ESC C-v, in the echo area 16

F

f 12
file, outputting to 4
filling 23
find-menu 11
first-node 8
footnotes, displaying 18
formatting without $\TeX$ 24
forward-char 6
forward-word 6
functions, describing 17

G

g 9
gc-compressed-files 20
get-help-window 17
get-info-help-node 17
getting started 3
global-next-node 8
global-prev-node 8
goto-node 9
grow-window 14

H

h 17
hard copy, simple 24
history-node 8

I

i 10
index-search 10
Info command summary, online 3
info file, selecting 4
info files, description of 1
INFO_PRINT_COMMAND, environment variable 17
isearch-backward 10
isearch-forward 10
ISO Latin characters 21
ISO-Latin 21

K

keep-one-window 14
keys, describing 17
kill-node 9

L

l 8
last-menu-item 11
last-node 8
learning Info 3
line breaking 23
list-visited-nodes 9

M

m 11
M-> 6
M-< 6
M-1 . . . M-9 18
M-b, in Info winows 6
M-b, in the echo area 15
M-d, in the echo area 15
M-DEL, in the echo area 15
M-f, in Info windows 6
M-f, in the echo area 15
M-r 6
M-TAB, in Info windows 12
M-TAB, in the echo area 15
M-v 7
M-y, in the echo area 16
makeinfo options 23
menu, following 5
menu-digit 11
menu-item 11
move-to-next-xref 12
move-to-prev-xref 12
move-to-window-line 6

N

n 8
next-index-match 10
next-line 6
next-node 8
next-window 13
node pointer defaults 26
node, selecting 4
nodes, description of 3
nodes, selection of 7
numeric arguments 17

O

options, makeinfo 23
outputting to a file 4

P

p 8
paragraphing 23
prev-line 6
prev-node 8
prev-window 13
print-node 17
printing 16
printing characters, in the echo area . . 15

Q

q 18
quit 18
quitting 18

R

r 12
redraw-display 7
RET, in Info windows 12
RET, in the echo area 15

S

s 9
screen, changing the height of 18
scroll-backward 7
scroll-behaviour 20
scroll-forward 6
scroll-other-window 14
scroll-step 21
scrolling 6
scrolling through node structure 7
search 9
searching 9
select-reference-this-line 12
select-visited-node 9
set-screen-height 18
set-variable 19
show-footnotes 18
show-index-match 20
single output file, forcing 24
SPC, in Info windows 6
SPC, in the echo area 16
split-window 13
splitting info files, avoiding 24

T

t 8
TAB, in Info windows 12
TAB, in the echo area 16
texinfo, description of. 1
tile-windows 14
tiling 14
toggle-wrap 7
top-node 8
tutorial for Info 3

U

u 8
universal-argument 17
up-node 8

V

valid info file 25
variables, describing 19
variables, setting 19
version information 5
view-file 9
visible-bell 19

W

where-is 17
windows, creating 13
windows, deleting 14
windows, manipulating 12
windows, selecting 13

X

xref-item 12

Rebuilding From Source

Cygnus Support Developer's Kit

Cygnus Support

Copyright © 1994, 1995 Cygnus Support

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

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

Table of Contents

Rebuilding From Source **1**

1 Configuration **3**
- 1.1 Options to `configure` 3
- 1.2 Specifying names for hosts and targets 5
  - Host names 5
  - Target names 6
  - `config.guess` 7
- 1.3 Source and Build Directories 8
- 1.4 Rebuilding using `gcc` 9

2 Compilation **11**

3 Installation **13**

4 Examples and suggestions **15**
- 4.1 The Heterogeneous Updateable Toolkit 15
  - How the HUT works 15
    - Configuring for the HUT 16
    - Setting links after installation 16
    - The Heterogeneous part 16
    - The Updateable part 17
  - Building and installing the HUT 17
- 4.2 How your Developer's Kit was built 18
- 4.3 Possible build variations 19
  - Building with the defaults 20
  - Example: setting *prefix* and *exec-prefix* 21
  - Example: different *srcdir* and *objdir* 23
  - Example: setting *prefix*/*exec-prefix* and *srcdir* ... 25
  - Multiple simultaneous builds 26

Rebuilding From Source

All Cygnus products are free software; your Developer's Kit includes complete source code for all programs. Because of this, you have the freedom to rebuild the tools for any of a variety of reasons, such as redefining compiled-in pathnames, or installing a source code patch from Cygnus.

Whatever the reason, we have designed and implemented an automatic configuration scheme to adapt the programs to different environments.

Rebuilding the programs from source requires these steps:

1. Configuration (see Chapter 1 "Configuration," page 3)
2. Compilation (see Chapter 2 "Compilation," page 11)
3. Installation (see Chapter 3 "Installation," page 13)

The first three chapters of this manual discuss the background behind each of the three steps involved in completely building and installing the Cygnus Support Developer's Kit. See Chapter 4 "Examples and suggestions," page 15, for variations on this theme. Follow the examples in "Building with the defaults," page 20, to recreate the distribution as it exists by default.

> *Note:* The procedures described in this note are for Unix systems. For DOS or LynxOS toolkits, contact Cygnus Support for instructions on rebuilding your Developer's Kit. In particular, since we do not provide a native DOS compiler, you must either rebuild on a Unix system or allow Cygnus to rebuild the toolkit for you.

We strongly recommend that you contact Cygnus Support if you wish to rebuild the Developer's Kit for cross-platform development. Contact your Cygnus technical representative, or get in touch with us at:

Cygnus Support
hotline: +1 415 903 1401
email: `support@cygnus.com`

Headquarters	*East Coast*
1937 Landings Drive	48 Grove St., Ste. 105
Mountain View, CA 94043 USA	Somerville, MA 02144 USA
+1 415 903 1400	+1 617 629 3000
+1 415 903 0122 fax	fax +1 617 629 3010

1 Configuration

The first and most important step in preparing source code to run on your system is to *configure* it so that, when built, the program exhibits the behavior you expect. The configuration process involves preparing a *Makefile* which contains default and/or customized information for your site and for your hardware/software system. If you are building the distribution for more than one platform, you must configure, compile, and install on each platform.

You can configure the software in this release by using the shell script `configure`. The shell script accepts one argument, the *host type*, although if you do not supply it `configure` is able to determine it under most circumstances (see "`config.guess`," page 7). In fact, in most cases we recommend you *not* specify the type of host.

There are also several possible options, including a '`--target=`' option to configure for cross-system development. For various examples of these options, see Chapter 4 "Examples and suggestions," page 15.

Your Developer's Kit contains full online documentation for Cygnus `configure`. See section "Using `configure`" in *Cygnus* `configure`, to read about `configure` in more detail, including information on how the `configure` options are related to '`Makefile`' variables.

1.1 Options to `configure`

This section summarizes the `configure` options and arguments used most often:

```
configure [ hosttype ]
          [ --prefix=dest ]
          [ --exec-prefix=bindest ]
          [ --srcdir=path ]
          [ --target=target ]
```

The '`--prefix`' and '`--exec-prefix`' options are particularly important. If you don't specify a *dest* or *bindest* directory, the '`Makefile`' installs binaries in subdirectories of '`/usr/cygnus/progressive-`*release*' (see Section 4.2 "How your Developer's Kit was built," page 18). These options are important because the *dest* and *bindest* directories are used for several purposes, most notably:

- *bindest* is the directory where binaries are installed.
- *bindest* is built into the compiler itself for the locations of GCC-specific include files, the locations of GCC subprograms, and the location of the GCC-specific library '`libgcc.a`'.
- *dest* is compiled into `info` as the default directory for the documentation.

See Section 4.1 "The Heterogeneous Updateable Toolkit," page 15, for hints on setting up your installation to be accessible and easily updated.

`hosttype` Configure the development tools to run on the specified *hosttype*. See "Host names," page 5. This argument is not usually required, since `configure` can automatically determine what kind of machine it runs on.

`--prefix=`*dest*
: *dest* is an installation directory *path prefix*, the root for the directories where `make install` installs programs, libraries, and other relevant files. After you configure with this option, '`make install install-info`' installs `info` files in '*dest*`/info`', MAN pages in '*dest*`/man`', and (unless you also use '`--exec-prefix`') binary programs in '*dest*`/bin`', and libraries in '*dest*`/lib`'. If you specify '`--prefix=/usr/local`', for example, `make install` puts the development tools in '`/usr/local/bin`'. (See Section 4.1 "The Heterogeneous Updateable Toolkit," page 15, for more detail.)

  Cygnus uses a *dest* of '`/usr/cygnus/progressive-`*date*'; see Section 4.2 "How your Developer's Kit was built," page 18. This is also the default *dest* for your source code. We recommend you always use the '`--prefix`' option to explicitly set the destination prefix.

`--exec-prefix=`*bindest*
: '`--exec-prefix`' serves the same purpose as '`--prefix`', but affects only machine-dependent binaries (programs and precompiled libraries). Specifying both '`--prefix`' and '`--exec-prefix`' allows you to segregate machine-dependent files, so that machine-independent files can be shared (see Section 4.1 "The Heterogeneous Updateable Toolkit," page 15).

  The default *bindest* is normally the value for `dest`, specified with '`--prefix`'. Cygnus specifies a *bindest* value of '`/usr/cygnus/progressive-`*date*`/H-`*hosttype*'; see Chapter 4 "Examples and suggestions," page 15. This is also the default for your source distribution, unless you set *dest* with '`--prefix`'. We recommend you always use the '`--exec-prefix`' option to explicitly set the machine-dependent destination prefix.

`--srcdir=`*path*
: Use this option to configure in directories separate from the source directories. `configure` writes configuration-specific files in the current directory, but arranges for them to use

the source in the directory *path*. `configure` creates directories under the working directory in parallel with the source directories below *path*. The default *path* is the directory in which `configure` resides; setting this option is redundant, but explicit.

Among other things, you can use this to build (or maintain) several configurations simultaneously, in separate *build directories*. See Section 1.3 "Source and Build Directories," page 8.

> *Warning:* This option is only supported if you use GNU Make.

`--target=`*target*

Configure the development tools for *cross-development* (compiling, debugging, or other processing) of programs running on the specified *target* (see "Target names," page 6). Without this option, the toolkit is configured as *native*, i.e., to manage programs that run on the same system as the development tools themselves.

Cross-development tools are named with a prefix of *target* in order to avoid confusion with the native tools. Thus, if the toolkit is built for a `mips-idt-ecoff` target, the compiler is named `mips-idt-ecoff-gcc`, the debugger is named `mips-idt-ecoff-gdb`, etc.

1.2 Specifying names for hosts and targets

Your tape is labeled to indicate the host (and target, if applicable) for which the binaries in the distribution are configured. The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, though the scheme is slightly different between hosts and targets.

Host names

The full naming scheme for hosts encodes three pieces of information in the following pattern:

architecture-vendor-os

For example, the full name for a Sun SPARCstation running SunOS 4.1.3 is

```
sparc-sun-sunos4.1.3
```

Remember that you can type `configure` without specifying a *`host-type`* and your host will be divined by `configure` (see "`config.guess`," page 7). In fact, we recommend this procedure on most systems.

> *Warning:* `configure` can represent a very large number of combinations of architecture, vendor, and operating system. There is by no means support for all possible combinations!

The following combinations refer to hosts supported by Cygnus Support. Some common short aliases are included, but these may be obsolete in the future. (For a matrix which shows all supported host/target combinations, see section "Overview" in *Release Notes*.)

canonical name	*alias*	*platform*
`sparc-sun-solaris2`	`sun4sol2`	Sun 4 running Solaris 2
`sparc-sun-sunos4.1.3`	`sun4`	Sun-4 running SunOS 4
`mips-dec-ultrix`	`decstation`	DECstation
`rs6000-ibm-aix`	`rs6000`	IBM RS6000
`mips-sgi-irix4`	`iris`	SGI Iris running Irix 4
`m68k-hp-hpux`	`hp300hpux`	HP 9000/300
`hppa1.1-hp-hpux`	`hp700`	HP 9000/700
`i386-unknown-sysv4`		UnixWare
`i386-lynx-lynxos`	`i386-lynx`	Intel *x*86 Lynxos 2.2
`m68k-lynx-lynxos`	`m68k-lynx`	Motorola 68k Lynxos 2.2
`sparc-lynx-lynxos`	`sparc-lynx`	SPARC Lynxos 2.2
`rs6000-lynx-lynxos`	`rs6000-lynx`	IBM RS6000 Lynxos 2.2
`alpha-dec-osf1.3`		DEC Alpha running OSF/1 v1.3

Target names

If you have a cross-development tape, the label also indicates the target for that configuration. The pattern for target names is

`architecture``[-`*`vendor`*`]-`*`objfmt`*

Target names differ slightly from host names in that the last variable indicates the object format rather than the operating system, and the second variable is often left out (this practice is becoming obsolete; in the future, all configuration names will be made up of three parts).

In cross-development configurations, each tool in the Developer's Kit is installed with the configured name of the target as a prefix. For example, if the C compiler is configured to generate COFF format code for the Motorola 680x0 family, the compiler is installed as '`m68k-coff-gcc`'.

> *Warning:* `configure` can represent a very large number of target name combinations of architecture, vendor, and object format. There is by no means support for all possible combinations!

This is a list of some of the more common targets supported by Cygnus Support. (Not all targets are supported on every host!) The list is not

exhaustive; see section "Overview" in *Release Notes*, for an up-to-date matrix which shows the host/target combinations supported by Cygnus.

Motorola 68000 family

`m68k-aout`	`a.out` object code format
`m68k-coff`	COFF object code format
`m68k-vxworks`	VxWorks environment
`m68k-lynx`	Lynxos 2.2 environment

Motorola 88000 family

`m88k-coff`	COFF object code format

Intel 960 family

`i960-vxworks5.0`	VxWorks environment (`b.out` format)
`i960-vxworks5.1`	VxWorks environment (COFF format
`i960-intel-nindy`	Nindy monitor

AMD 29000 family

`a29k-amd-udi`	UDI monitor interface

To use the minimon *interface, use this configuration with the auxiliary program* `MONTIP`*, available from AMD.*

SPARC family

`sparc-vxworks`	VxWorks environment
`sparc-aout`	`a.out` object code format
`sparclite-aout`	`a.out` object code format
`sparclite-coff`	COFF object code format

Intel 80X86 family

`i386-aout`	`a.out` object code format
`i386-netware`	NetWare NLM
`i386-lynx`	LynxOS 2.2 environment

IDT/MIPS R3000

`mips-idt-ecoff`	IDT R3000, big endian ECOFF
`mipsel-idt-ecoff`	IDT R3000, little endian ECOFF
`mips64-idt-ecoff`	IDT R4000 ECOFF

Hitachi H8300

`h8300-hms-coff`	COFF object code format

Hitachi SH

`sh-hms-coff`	COFF object code format

Z8000

`z8k-coff`	COFF object code format

`config.guess`

`config.guess` is a shell script which attempts to deduce the host type from which it is called, using system commands like `uname` if they are available. `config.guess` is remarkably adept at deciphering the proper configuration for your host; if you are building a tree to run on

the same host on which you're building it, we recommend *not* specifying the `hosttype` argument.

`config.guess` is called by `configure`; you need never run it by hand, unless you're curious about the output.

1.3 Source and Build Directories

Builds are most often done in the same directory where the source lies. However, if you don't have enough disk space there, or if you wish to compile the Developer's Kit for more than one configuration, you may find it easiest to configure and build in a different directory from the source.

To build in a location different from the source directory, first create the build directory, which we'll call '*objdir*':

```
$ mkdir objdir
$ cd objdir
```

Then run `configure` from the top level of the source directory, which we'll call '*srcdir*'. You don't need to specify the '`--srcdir=path`' option to `configure` (see Section 1.1 "configure options," page 3), but we show it here for the purposes of the example:

```
$ srcdir/configure --srcdir=srcdir . . .
```

The default for *srcdir* is the directory in which `configure` resides.

`configure` creates a '`Makefile`' in the current directory, '*objdir*'. When you run `make` here, object files are created in '*objdir*' from the source code in '*srcdir*'. For example (assume source code is in '`/usr/local/src`', and binaries are to be installed under '`/usr/local`'):

```
$ mkdir /usr/local/obj/sun4
$ cd /usr/local/obj/sun4
$ /usr/local/src/configure --srcdir=/usr/local/src
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
Created "Makefile" in /usr/local/obj/sun4. . .
```

This is extremely useful if you need to create more than one installation of the Developer's Kit. For example, if you wish to rebuild and install the toolkit for a given host as well as a cross-development system for the same host, you can use a different build directory for each toolkit, with different options to `configure` for each build. In this way, the object files for each configuration exist simultaneously but independently, even if they are meant to install finally into the same repository.

See Section 4.3 "Possible build variations," page 19, for examples of the build process using separate directories for source and object files.

Again, please contact Cygnus Support if you have any trouble.

1.4 Rebuilding using `gcc`

If you've built and installed the compiler in a *native* configuration, you may wish to use it to rebuild itself. To do this, set the environment variable `CC` to the installed version of `gcc`, and reconfigure and rebuild the toolkit.

```
$ CC=installed-dir/gcc configure ...
$ make CC=installed-dir/gcc ...
```

For example, if you installed the Developer's Kit in '`/usr/local`', use the following commands to rebuild with the installed GCC:

```
$ CC=/usr/local/bin/gcc configure ...
$ make CC=/usr/local/bin/gcc ...
```

Make sure you specify the same compiler for `CC` for both the `configure` and `make` steps.

(Note: These examples assumes you are using a Bourne-compatible shell (`sh`, `bash`, `ksh`); contact Cygnus Support if you encounter any problems.)

2 Compilation

After you've run `configure`, compilation is straightforward. To compile all the programs in the Developer's Kit, run:

```
make all info > make.log
```

The examples suggest capturing the `make` output in a '`make.log`' file, because the output is lengthy.

`make` creates a set of binaries which run on your *hosttype* and which compile code for the specified *target* (or for the *hosttype* in a native configuration). Programs which require compiled-in pathnames are built with the values you specified on the command line to `configure` with the options '`--prefix`' and '`--exec-prefix`'. In cross-development configurations, programs are named with *target* as a prefix; for example, the cross-compiler is named *target*`-gcc`.

The overall '`Makefile`' propagates the value of the `CC` variable explicitly, so that you can easily control the compiler used in this step. `CFLAGS` is treated the same way. For instance, to build the compiler a second time, using `gcc` to compile itself (after building and installing it in the alternate directory '`/usr/local`'; see Section 1.4 "Rebuilding using `gcc`," page 9), you might follow this example:

```
$ CC=/usr/local/bin/gcc configure ...
$ make CC=/usr/local/bin/gcc CFLAGS=-O2 all info > make.log
```

Make sure you specify the same compiler for `CC` for both the `configure` and `make` steps.

The conventional targets `all`, `install`, and `clean` are supported at all levels of '`Makefile`'. (Other targets are supported as well, as appropriate in each directory.) Each '`Makefile`' in the source directories includes ample comments to help you read it. If you are not familiar with `make`, refer to section "Overview of `make`" in *GNU Make: A Program for Directing Recompilation*.

3 Installation

Once the software is compiled, installation is elementary. Simply type:

```
make install install-info >> make.log
```

`make` installs the Developer's Kit into the locations you specified on the `configure` command line with '`--prefix`' and '`--exec-prefix`'.

The Cygnus installation process calls for a few links to be created after installation, though if you've specified values of '`--prefix`' and/or '`--exec-prefix`' different from the defaults, your links may be different or even unnecessary. See Section 4.2 "How your Developer's Kit was built," page 18, for more information on these links.

See Section 4.2 "How your Developer's Kit was built," page 18, and the *Installation Notes*, for more discussion on installing your Developer's Kit.

4 Examples and suggestions

Following are several examples of the build process, and a few suggestions on rebuilding and installing your Cygnus Support Developer's Toolkit.

The most useful suggestion we can offer is a process by which you can make your Developer's Kit both easily updateable and easily accessible to your developers, even in a heterogeneous network environment. We use this process to build the binary distribution of the Developer's Kit.

4.1 The Heterogeneous Updateable Toolkit

The process described here combines the use of the '`--prefix`' and '`--exec-prefix`' options to `configure` (see Section 1.1 "`configure` options," page 3) with the creation of strategic symbolic links in the installation directory. See Section 4.2 "How your Developer's Kit was built," page 18, for an example of this process.

In the following examples, the variable *`release`* refers to a given Cygnus Support release number, such as '`progressive-94q4`' (many of the examples simply use '`94q4`'). The variable *`hosttype`* refers to the configure name for the given host type, such as '`sparc-sun-sunos4.1.3`' (see Section 1.2 "Specifying names for hosts and targets," page 5). These values are on the distribution tape label.

In order for the Developer's Kit to be accessible in a heterogeneous network environment, it must be configured, compiled, and installed on each *host type* (one installation for the Sun4s on the network, one for the DECstations, etc.), using the configuration paradigm discussed below (for a more general discussion, see "Building and installing the HUT," page 17). The symbolic links must also be set up on each host.

How the HUT works

The Heterogeneous Updateable Toolkit depends on the use of the '`--prefix`' and '`--exec-prefix`' options to `configure`, as well as to two strategic symbolic links in the installation directory.

See "Building and installing the HUT," page 17, and Section 4.3 "Possible build variations," page 19, for more general examples and suggestions.

Configuring for the HUT

The binary distribution on your tape is configured with the following command:

```
configure --prefix=/usr/cygnus/progressive-release \
   --exec-prefix=/usr/cygnus/progressive-release/H-hosttype \
  [ --target=target ]
```

This configuration is used for each *hosttype* for a given *release*. The '--target=*target*' option is only needed for cross-development configurations. If you are building installations for more than one platform, each platform should follow this standard. This configuration places machine-independent files (like documentation and library sources) in

```
/usr/cygnus/progressive-release
```

and machine-dependent files (like binary programs and precompiled library archives) in

```
/usr/cygnus/progressive-release/H-hosttype
```

Using these installation locations allows us to create multiple Heterogeneous Updateable Toolkits from a single source tree by simply putting two strategic symbolic links in place.

Setting links after installation

We recommend setting two links after building the Heterogeneous Updateable Toolkit. One makes the toolkit easily updateable; the other makes the tools available in a heterogeneous network environment. *Note:* If you choose not to set the links in place, you must set your PATH according to the value you choose for *exec-prefix*.

The Heterogeneous part

The installed distribution is useful in a heterogeneous environment due to the properties of local public installation directories like '/usr'. '/usr' is local to each machine, even machines of the same architecture. However, subdirectories of '/usr' can be symbolic links to other locations on disks shared throughout the network.

The HUT creation process (and the Cygnus progressive installation process) suggests the symbolic link

```
ln -s /usr/cygnus/progressive/H-hosttype /usr/progressive
```

on each host for which the toolkit is installed. This way, users on each system have access to the Developer's Kit through '/usr/progressive/bin', since '/usr/progressive' on each *hosttype* points to the binaries for that *hosttype*.

The Updateable part

The installed distribution is easily updated via the symbolic link

```
ln -s /usr/cygnus/progressive-release /usr/cygnus/progressive
```

Each time you install a new `release` with a similar configuration, you only need to change this symbolic link to give your users access to the new version of the toolkit. You can also revert the change by switching the symbolic link back to the previous `release` if needed, as the previous `release` is also still available. Once you decide to keep the new `release`, you can remove the old one to conserve disk space.

Building and installing the HUT

If you wish to rebuild your Developer's Kit with the same precompiled pathnames as the default (under '`/usr/cygnus`'), simply follow the example in "Building with the defaults," page 20.

Otherwise, decide on an installation directory; for the purposes of these examples, we'll call it '*instdir*'. Run `configure`, specifying options so that the host-dependent files described by '`--exec-prefix`' reside in a level underneath the host-independent files designated by '`--prefix`', and so that both host-dependent and host-independent files are designated with the release number *release*.

This division allows the distribution to be both easily updated and easily accessed after installation (see "How the HUT works," page 15).

```
$ configure --prefix=instdir/release \
  --exec-prefix=instdir/release/H-hosttype
```

You can also build in an *object directory*, different from that which holds the sources, allowing more than one compiled tree to be available simultaneously.

```
$ mkdir objdir1     (this one is native)
$ cd objdir1
$ srcdir/configure --prefix=instdir/release \
    --exec-prefix=instdir/release/H-hosttype

$ mkdir objdir2     (this one is for cross-development)
$ cd objdir2
$ srcdir/configure --prefix=instdir/release \
    --exec-prefix=instdir/release/H-hosttype \
    --target=target
```

Once the configuration is set, compilation is straightforward:

```
$ make all info >> make.log
```

Installation is straightforward as well (the example shows access to root; this is usually, though certainly not always, needed to install into publicly accessible places like '`/usr`'):

```
$ su
# make install install-info >> make.log
```

The final process is to set links in place, so the toolkit is easily accessible and updateable, and available in a heterogeneous environment. *pub*, shown below, indicates a top-level publicly accessible directory, such as '/usr'. *rel* is a truncated version of *release*, meant to be more general; if *release* is 'progressive-94q4', *rel* might be 'progressive'.

```
# ln -s instdir/release instdir/rel
# ln -s instdir/rel/H-hosttype pub/rel
# exit     (root access not needed beyond this)
```

Now, anyone who puts '*pub*/*rel*' in her or his path has full access to the installed tools. You can also build and install the tools for other host types; these other toolkits are available from the "same" location, '*pub*/*rel*', because *pub* is local to each machine. (For more discussion of these links, see "How the HUT works," page 15.)

For concrete examples of this process, see Section 4.3 "Possible build variations," page 19.

4.2 How your Developer's Kit was built

The files in your Developer's Kit distribution are compiled with the following options to configure (the line is only split so that it fits on the printed page; it is intended as a single command line):

```
configure --prefix=/usr/cygnus/progressive-release \
 --exec-prefix=/usr/cygnus/progressive-release/H-hosttype
```

where *release* indicates the Cygnus release number for this distribution, e.g., '94q4', and *hosttype* indicates the architecture and operating system configuration on which this software is to run, e.g., 'sparc-sun-sunos4.1.3' (see "Host names," page 5). Cross-development distributions are configured with

```
configure --prefix=/usr/cygnus/progressive-release \
 --exec-prefix=/usr/cygnus/progressive-release/H-hosttype \
 --target=target
```

where *target* indicates the architecture and object file format for which the Developer's Kit is to generate code (see "Target names," page 6).

In other words, in both native and cross-development configurations, host-independent files (text files, library source code, etc.) reside by default in

```
/usr/cygnus/progressive-release
```

while host-dependent files, like precompiled libraries and the tools themselves, reside by default in

```
/usr/cygnus/progressive-release/H-hosttype
```

For example, the Sun 4 cross 'm68k-aout' cross-compiler for the Progressive 94q4 release was configured with

```
configure --prefix=/usr/cygnus/progressive-94q4 \
 --exec-prefix=/usr/cygnus/progressive-94q4/H-sparc-sun-sunos4.1.3 \
 --target=m68k-aout
```

Installation

The installation procedure directs you to run the `Install` script and then set some links in place so that the distribution may be easily accessed and updated (see section "Installing your Developer's Kit" in *Installation Notes*) as follows:

```
ln -s /usr/cygnus/progressive-release /usr/cygnus/progressive
ln -s /usr/cygnus/progressive/H-hosttype /usr/progressive
```

This combination of a separate links provide the following benefits:

- Access is granted to anyone who puts '`/usr/progressive/bin`' in her or his path
- Updating is simplified; when you install a new Developer's Kit (with a different value for *release*), you only need to change the link '`/usr/cygnus/progressive`' to bring the new toolkit online
- Installed toolkits for multiple platforms can coexist in the same location and share host-independent files, conserving disk space and easing maintenance; if you install the toolkit for more than one platform, '`/usr/progressive`' on each platform points to the host-dependent files for that platform, yet the host-independent files are shared (see Section 4.1 "The Heterogeneous Updateable Toolkit," page 15)

For more details on installing the default binary distribution from Cygnus Support, see section "Installing your Developer's Kit" in *Installation Notes*.

4.3 Possible build variations

There are several permutations of variations in building the Cygnus Support Developer's Kit. In this section we try to discuss the majority of the possibilities. Feel free to contact Cygnus Support with any questions or problems.

In these examples, we assume the source code has been extracted from the tape into '`/usr/local/src`' on a Sun 4 running SunOS 4.1.3. Examples are shown using a Bourne-compatible shell (`sh`, `bash`, `ksh`). Sample command lines which are too long to fit on the printed page are wrapped as follows:

```
$ ./configure --prefix=/usr/local/94q4 \
    --exec-prefix=/usr/local/94q4/H-sun4 \
    --srcdir=/usr/local/src
```

Such lines are intended to be typed on one single command line.

Examples show a *release* number of '94q4', and (often) a *hosttype* of sun4. In practice, we recommend the values 'progressive-94q4' and 'sparc-sun-sunos4.1.3', respectively; see Section 4.2 "How your Developer's Kit was built," page 18. The values are shortened here to make the examples fit on the printed page.

The examples also show the creation of two links. The links aren't necessary, but they do provide an easy way to make your Developer's Kit easily updateable and accessible in a heterogeneous environment.

The first is a link to make updating easier, so that when you upgrade the Developer's Kit you only need to switch that link to bring it online. The second link allows the creation of several toolkits in a heterogeneous network; each different host can have a link to the toolkit configured for the host's architecture and operating system. See Section 4.1 "The Heterogeneous Updateable Toolkit," page 15.

Building with the defaults

A default build is one in which the configure defaults are used exclusively. The software is built in the same directory as the source code.

The defaults are as follows (the *release* is shown as '94q4'):

prefix	/usr/cygnus/progressive-94q4
exec-prefix	/usr/cygnus/progressive-94q4/H-*hosttype* *(we show the* hosttype *as* 'sparc-sun-sunos4.1.3'*)*
target	*nonexistent by default; cross example shows* 'm68k-coff'
srcdir	*current directory; example shows* '/usr/local/src'
objdir	*same as* srcdir
release	*shown on distribution tape (example shows* '94q4'*)*

The native and cross examples show a complete walk-through for each type of build. (The only difference is the use of the '--target' option to configure for a cross-development toolkit. We'll use 'm68k-coff' as an example for building a cross-development toolkit.) The programs are to be accessible after installation via the symbolic link '/usr/progressive' (see Section 4.1 "The Heterogeneous Updateable Toolkit," page 15). Builds are shown independently; for an example of a simultaneous build, see "Multiple simultaneous builds," page 26.

Native:

```
$ cd /usr/local/src
$ ./configure
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
```

```
$ make all info > ./make.log
. . . time passes . . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/cygnus/progressive-94q4
# make install install-info >> ./make.log
# ln -s /usr/cygnus/progressive-94q4 /usr/cygnus/progressive
# ln -s /usr/cygnus/progressive/H-sparc-sun-sunos4.1.3 /usr/progressive
# exit
$ ls /usr/progressive/bin
ar              gcov            objdump
as              gdb             patch
byacc           genclass        ranlib
c++             gperf           sdiff
c++filt         gprof           send-pr
cmp             info            size
diff            install-sid     sparc-sun-sunos4.1.3-gcc
diff3           ld              strings
flex            make            strip
g++             makeinfo        texi2dvi
gasp            nm              texindex
gcc             objcopy
```

Cross:

```
$ cd /usr/local/src
$ ./configure --target=m68k-coff
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes . . .
$ make all info > ./make.log
. . . time passes . . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/cygnus/progressive-94q4
# make install install-info >> ./make.log
# ln -s /usr/cygnus/progressive-94q4 /usr/cygnus/progressive
# ln -s /usr/cygnus/progressive/H-sparc-sun-sunos4.1.3 /usr/progressive
# exit
$ ls /usr/progressive/bin
byacc     install-sid        m68k-coff-gdb      m68k-coff-strip
cmp       m68k-coff-ar       m68k-coff-ld       make
diff      m68k-coff-as       m68k-coff-nm       makeinfo
diff3     m68k-coff-c++      m68k-coff-objcopy  patch
flex      m68k-coff-c++filt  m68k-coff-objdump  sdiff
gcov      m68k-coff-g++      m68k-coff-ranlib   send-pr
genclass  m68k-coff-gasp     m68k-coff-size     texi2dvi
info      m68k-coff-gcc      m68k-coff-strings  texindex
```

In either of the above examples, you must set your `PATH` to include '`/usr/progressive/bin`' in order to access the tools easily.

Example: setting *prefix* and *exec-prefix*

Use '`--prefix`' and '`--exec-prefix`' to explicitly set installation directories. These variables are set to subdirectories of

'/usr/cygnus/progressive-*date*' by default (see "Building with the defaults," page 20). For more involved discussion on the nuances of these options, see Section 4.1 "The Heterogeneous Updateable Toolkit," page 15. For discussion on the options themselves, see Section 1.1 "Options to configure," page 3.

This example shows different installation directories from the default. The defaults for this example are as follows (the *release* is shown as '94q4'):

prefix	*set on command line*
exec-prefix	*set on command line*
target	*nonexistent by default; cross example shows* 'm68k-coff'
srcdir	*current directory; example shows* '/usr/local/src'
objdir	*same as* srcdir
release	*shown on distribution tape (example uses* '94q4'*)*

The native and cross examples show a complete walk-through for each type of build. (The only difference is the use of the '--target' option to configure for a cross-development toolkit. We'll use 'm68k-coff' as an example for building a cross-development toolkit.)

For this example, we'll set the configuration to install host-independent files (documentation, library source code) in '/usr/local', and host-dependent files (binary programs, precompiled libraries) in '/usr/local/H-sun4'. The programs are to be accessible after installation via the symbolic link '/usr/progressive' (see Section 4.1 "The Heterogeneous Updateable Toolkit," page 15). Builds are shown independently; for an example of a simultaneous build, see "Multiple simultaneous builds," page 26.

Native:

```
$ cd /usr/local/src
$ ./configure --prefix=/usr/local/94q4 \
    --exec-prefix=/usr/local/94q4/H-sun4
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/local/94q4
# make install install-info >> ./make.log
# ln -s /usr/local/94q4 /usr/local/progressive
# ln -s /usr/local/progressive/H-sun4 /usr/progressive
# exit
$ ls /usr/progressive/bin
ar              gcov            objdump
as              gdb             patch
byacc           genclass        ranlib
c++             gperf           sdiff
c++filt         gprof           send-pr
cmp             info            size
```

```
diff            install-sid     sparc-sun-sunos4.1.3-gcc
diff3           ld              strings
flex            make            strip
g++             makeinfo        texi2dvi
gasp            nm              texindex
gcc             objcopy
```

Cross:

```
$ cd /usr/local/src
$ ./configure --target=m68k-coff --prefix=/usr/local/94q4 \
    --exec-prefix=/usr/local/94q4/H-sun4
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su         (may need root priviledge to install in '/usr')
# mkdir /usr/local/94q4
# make install install-info >> ./make.log
# ln -s /usr/local/94q4 /usr/local/progressive
# ln -s /usr/local/progressive/H-sun4 /usr/progressive
# exit
$ ls /usr/progressive/bin
byacc     install-sid        m68k-coff-gdb      m68k-coff-strip
cmp       m68k-coff-ar       m68k-coff-ld       make
diff      m68k-coff-as       m68k-coff-nm       makeinfo
diff3     m68k-coff-c++      m68k-coff-objcopy  patch
flex      m68k-coff-c++filt  m68k-coff-objdump  sdiff
gcov      m68k-coff-g++      m68k-coff-ranlib   send-pr
genclass  m68k-coff-gasp     m68k-coff-size     texi2dvi
info      m68k-coff-gcc      m68k-coff-strings  texindex
```

In either of the above examples, you must set your `PATH` to include '`/usr/progressive/bin`' in order to access the tools easily.

Example: different *srcdir* and *objdir*

The concept of different source and build directories comes from the practice of building the same toolkit for several different platforms, in a heterogeneous environment. It is often convenient to keep object files separate from the sources from which they were derived. For more discussion, see Section 1.3 "Source and Build Directories," page 8.

This example shows different source and object directories only. The defaults for this example are as follows (the *release* is shown as '`94q4`'):

```
prefix       /usr/cygnus/progressive-94q4
exec-prefix  /usr/cygnus/progressive-94q4/H-hosttype
                 (we show the hosttype as 'sparc-sun-sunos4.1.3')
target       nonexistent by default; cross example uses 'm68k-coff'
srcdir       example shows '/usr/local/src'
objdir       example shows '/usr/local/obj/native' and
               '/usr/local/obj/m68k-coff'
```

`release` *shown on distribution tape (example uses '*`94q4`*')*

The native and cross examples show a complete walk-through for each type of build. (The only difference is the use of the '`--target`' option to `configure` for a cross-development toolkit. We'll use '`m68k-coff`' as an example for building a cross-development toolkit.)

Note: The use of '`--srcdir`' is redundant, as the default source directory is the one in which `configure` itself resides. We show it here for the purposes of the example. Builds are shown independently; for an example of a simultaneous build, see "Multiple simultaneous builds," page 26. *This option is only supported when you use* GNU `make`*.*

Native:

```
$ cd /usr/local/obj/native
$ /usr/local/src/configure --srcdir=/usr/local/src
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/cygnus/progressive-94q4
# make install install-info >> ./make.log
# ln -s /usr/cygnus/progressive-94q4 /usr/cygnus/progressive
# ln -s /usr/cygnus/progressive/H-sparc-sun-sunos4.1.3 /usr/progressive
# exit
$ ls /usr/progressive/bin
ar              gcov            objdump
as              gdb             patch
byacc           genclass        ranlib
c++             gperf           sdiff
c++filt         gprof           send-pr
cmp             info            size
diff            install-sid     sparc-sun-sunos4.1.3-gcc
diff3           ld              strings
flex            make            strip
g++             makeinfo        texi2dvi
gasp            nm              texindex
gcc             objcopy
```

Cross:

```
$ cd /usr/local/obj/m68k-coff
$ /usr/local/src/configure --target=m68k-coff \
    --srcdir=/usr/local/src
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/cygnus/progressive-94q4
# make install install-info >> ./make.log
# ln -s /usr/cygnus/progressive-94q4 /usr/cygnus/progressive
# ln -s /usr/cygnus/progressive/H-sparc-sun-sunos4.1.3 /usr/progressive
```

```
# exit
$ ls /usr/progressive/bin
byacc     install-sid        m68k-coff-gdb      m68k-coff-strip
cmp       m68k-coff-ar       m68k-coff-ld       make
diff      m68k-coff-as       m68k-coff-nm       makeinfo
diff3     m68k-coff-c++      m68k-coff-objcopy  patch
flex      m68k-coff-c++filt  m68k-coff-objdump  sdiff
gcov      m68k-coff-g++      m68k-coff-ranlib   send-pr
genclass  m68k-coff-gasp     m68k-coff-size     texi2dvi
info      m68k-coff-gcc      m68k-coff-strings  texindex
```

In either of the above examples, you must set your `PATH` to include '`/usr/progressive/bin`' in order to access the tools easily.

Example: setting *prefix*/*exec-prefix* and *srcdir*

This example shows different source and object directories, as well as different installation directories from the default. The defaults for this example are as follows (the *release* is shown as '`94q4`'):

`prefix`	*set on command line*
`exec-prefix`	*set on command line* (*we show the* `hosttype` *as* '`sparc-sun-sunos4.1.3`')
`target`	*nonexistent by default; cross example uses* '`m68k-coff`'
`srcdir`	*example shows* '`/usr/local/src`'
`objdir`	*example shows* '`/usr/local/obj/native`' *and* '`/usr/local/obj/m68k-coff`'
`release`	*shown on distribution tape (example uses* '`94q4`'*)*

The native and cross examples show a complete walk-through for each type of build. (The only difference is the use of the '`--target`' option to `configure` for a cross-development toolkit. We'll use '`m68k-coff`' as an example for building a cross-development toolkit.)

Note: The use of '`--srcdir`' is redundant, as the default source directory is the one in which `configure` itself resides. We show it here for the purposes of the example. Builds are shown independently; for an example of a simultaneous build, see "Multiple simultaneous builds," page 26. *This option is only supported when you use* GNU `make`.

Native:

```
$ cd /usr/local/obj/native
$ /usr/local/src/configure --srcdir=/usr/local/src \
    --prefix=/usr/local/94q4 --exec-prefix=/usr/local/94q4/H-sun4
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/local/94q4
# make install install-info >> ./make.log
# ln -s /usr/local/94q4 /usr/local/progressive
```

```
# ln -s /usr/local/94q4/H-sun4 /usr/progressive
# exit
$ ls /usr/progressive/bin
ar              gcov            objdump
as              gdb             patch
byacc           genclass        ranlib
c++             gperf           sdiff
c++filt         gprof           send-pr
cmp             info            size
diff            install-sid     sparc-sun-sunos4.1.3-gcc
diff3           ld              strings
flex            make            strip
g++             makeinfo        texi2dvi
gasp            nm              texindex
gcc             objcopy
```

Cross:

```
$ cd /usr/local/obj/m68k-coff
$ /usr/local/src/configure --target=m68k-coff --srcdir=/usr/local/src \
    --prefix=/usr/local/94q4 --exec-prefix=/usr/local/94q4/H-sun4
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# mkdir /usr/local/94q4
# make install install-info >> ./make.log
# ln -s /usr/local/94q4 /usr/local/progressive
# ln -s /usr/local/progressive/H-sun4 /usr/progressive
# exit
$ ls /usr/progressive/bin
byacc     install-sid        m68k-coff-gdb      m68k-coff-strip
cmp       m68k-coff-ar       m68k-coff-ld       make
diff      m68k-coff-as       m68k-coff-nm       makeinfo
diff3     m68k-coff-c++      m68k-coff-objcopy  patch
flex      m68k-coff-c++filt  m68k-coff-objdump  sdiff
gcov      m68k-coff-g++      m68k-coff-ranlib   send-pr
genclass  m68k-coff-gasp     m68k-coff-size     texi2dvi
info      m68k-coff-gcc      m68k-coff-strings  texindex
```

In either of the above examples, you must set your `PATH` environment variable to include '`/usr/progressive/bin`' in order to access the tools easily.

Multiple simultaneous builds

If the source code for your distribution resides on a disk shared by other machines in the network, you can build for all host types simultaneously by using different *build directories*. See Section 1.3 "Source and Build Directories," page 8, for details on building with different source

and build directories; also see "Example: different *srcdir* and *objdir*," page 23, for an example.

Multiple simultaneous builds can be conducted on the same source code simply by using different build directories. For example, assume we have source code in '/usr/local/src', and wish to build a native toolkit for our Sun SPARCstation running SunOS 4.1.3 (sparky) and our DECstation running Ultrix (deckard), and that we also wish to build a cross-development toolkit for each host for a 'm68k-aout' target.

> *Note:* this example shows a complete build for all four configurations, one native development system and one cross-development system for each of two hosts, including installation and links. See Section 4.1 "The Heterogeneous Updateable Toolkit," page 15, for more general discussions and examples.

All of these toolkits are to be installed into

```
/usr/local/progressive-94q4
```

(The cross-development tools are installed with the native tools; however, the cross tools have a prefix of *target*, e.g., 'm68k-aout-gcc'.) They are to be linked so that '/usr/progressive/bin' on each host points toward the correct binaries for that host:

```
/usr/local/progressive-94q4/H-hosttype/bin
```

but the machine-independent files in

```
/usr/local/progressive-94q4
```

are shared across platforms. We accomplish this by actually installing into a shared disk called 'shared', and creating links from '/usr/local'.

first, set up the shared space in '/shared'

```
$ mkdir /shared/local
$ su      (may need root priviledge to put link in '/usr')
# ln -s /shared/local /usr/local
# mkdir /usr/local/progressive-94q4
# exit
```

now we build the native toolset for the Sun

```
$ uname -a
SunOS sparky 4.1.3_U1 1 sun4m
$ mkdir /usr/local/obj/sun4native
$ cd /usr/local/obj/sun4native
$ /usr/local/src/configure --prefix=/usr/local/progressive-94q4 \
    --exec-prefix=/usr/local/progressive-94q4/H-sun4 \
    --srcdir=/usr/local/src
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
```

```
$ su          (may need root priviledge to install in '/usr')
# make install install-info >> ./make.log
# exit

        (now the cross toolkit)

$ mkdir /usr/local/obj/sun4-x-m68k
$ cd /usr/local/obj/sun4-x-m68k
$ /usr/local/src/configure --prefix=/usr/local/progressive-94q4 \
    --exec-prefix=/usr/local/progressive-94q4/H-sun4 \
    --srcdir=/usr/local/src --target=m68k-aout
Configuring for a sparc-sun-sunos4.1.3_U1 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# make install install-info >> ./make.log

        now create the links which bring the toolkit online

# ln -s /usr/local/progressive-94q4 /usr/local/progressive
# ln -s /usr/local/progressive/H-sun4 /usr/progressive
# exit
```

'`/usr/progressive`*' on* `sparky` *now points to the Sun4-specific installation*

'`/shared`*' already exists; now we build the native toolset for the DECstation*

```
$ rlogin deckard
$ uname -a
ULTRIX deckard 4.2 0 RISC
$ mkdir /usr/local/obj/decnative
$ cd /usr/local/obj/decnative
$ /usr/local/src/configure --prefix=/usr/local/progressive-94q4 \
    --exec-prefix=/usr/local/progressive-94q4/H-decstn \
    --srcdir=/usr/local/src
Configuring for a mips-dec-ultrix4.2 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
      remember, '/usr/local/progressive-94q4' already exists
# make install install-info >> ./make.log
# exit
```

(continued on next page. . .)

(now the cross toolkit)

```
$ mkdir /usr/local/obj/sun4cross
$ cd /usr/local/obj/sun4native
$ /usr/local/src/configure --prefix=/usr/local/progressive-94q4 \
    --exec-prefix=/usr/local/progressive-94q4/H-decstn \
    --srcdir=/usr/local/src --target=m68k-aout
Configuring for a mips-dec-ultrix4.2 host.
. . . time passes. . .
$ make all info > ./make.log
. . . time passes. . .
$ su          (may need root priviledge to install in '/usr')
# make install install-info >> ./make.log
```

now create the link which brings the toolkit online; remember, the other link
`/usr/local/progressive -> /usr/local/progressive-94q4`
already exists from the Sun build

```
# ln -s /usr/local/progressive/H-decstn /usr/progressive
# exit
```

'`/usr/progressive`' on `deckard` now points to the DEC-specific installation

(This example shows a Bourne-compatible shell (`sh`, `bash`, `ksh`); contact Cygnus if you have any trouble.)

'`/usr/progressive/bin`' on each host now points toward the binaries built for that *`hosttype`*, while the machine-independent files in '`/usr/local/progressive`' are shared between the two builds. In addition, each installation contains both a native development toolkit and a cross-development toolkit for the '`m68k-aout`' target.

Please contact Cygnus Support if any of this is unclear, or if you have any questions.

Comparing and Merging Files

diff, diff3, sdiff, cmp, and patch
Edition 1.3, for diff 2.7 and patch 2.1
September 1993

by David MacKenzie, Paul Eggert, and Richard Stallman

Copyright © 1992, 1993, 1994, 1995 Free Software Foundation, Inc.

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided 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 manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

Short Contents

Overview 1
1 What Comparison Means 3
2 `diff` Output Formats 9
3 Comparing Directories 29
4 Making `diff` Output Prettier 31
5 `diff` Performance Tradeoffs 33
6 Comparing Three Files 35
7 Merging From a Common Ancestor 39
8 Interactive Merging with `sdiff` 45
9 Merging with `patch` 47
10 Tips for Making Patch Distributions 53
11 Invoking `cmp` 55
12 Invoking `diff` 57
13 Invoking `diff3` 65
14 Invoking `patch` 69
15 Invoking `sdiff` 77
16 Incomplete Lines 81
17 Future Projects 83
Concept Index 87

Table of Contents

Overview **1**

1 What Comparison Means **3**
- 1.1 Hunks 3
- 1.2 Suppressing Differences in Blank and Tab Spacing 4
- 1.3 Suppressing Differences in Blank Lines 5
- 1.4 Suppressing Case Differences 5
- 1.5 Suppressing Lines Matching a Regular Expression 5
- 1.6 Summarizing Which Files Differ 6
- 1.7 Binary Files and Forcing Text Comparisons 6

2 `diff` Output Formats **9**
- 2.1 Two Sample Input Files 9
- 2.2 Showing Differences Without Context 9
  - 2.2.1 Detailed Description of Normal Format 10
  - 2.2.2 An Example of Normal Format 11
- 2.3 Showing Differences in Their Context 11
  - 2.3.1 Context Format 11
    - 2.3.1.1 Detailed Description of Context Format 12
    - 2.3.1.2 An Example of Context Format 12
    - 2.3.1.3 An Example of Context Format with Less Context 13
  - 2.3.2 Unified Format 14
    - 2.3.2.1 Detailed Description of Unified Format 14
    - 2.3.2.2 An Example of Unified Format 14
  - 2.3.3 Showing Which Sections Differences Are in 15
    - 2.3.3.1 Showing Lines That Match Regular Expressions 15
    - 2.3.3.2 Showing C Function Headings 16
  - 2.3.4 Showing Alternate File Names 16
- 2.4 Showing Differences Side by Side 16
- 2.5 Controlling Side by Side Format 17
  - 2.5.1 An Example of Side by Side Format 18
- 2.6 Making Edit Scripts 18
  - 2.6.1 `ed` Scripts 18
    - 2.6.1.1 Detailed Description of `ed` Format 19
    - 2.6.1.2 Example `ed` Script 19
  - 2.6.2 Forward `ed` Scripts 20
  - 2.6.3 RCS Scripts 20

2.7 Merging Files with If-then-else 21
2.7.1 Line Group Formats 21
2.7.2 Line Formats 24
2.7.3 Detailed Description of If-then-else Format 26
2.7.4 An Example of If-then-else Format 27

3 Comparing Directories 29

4 Making `diff` Output Prettier 31
4.1 Preserving Tabstop Alignment 31
4.2 Paginating `diff` Output 31

5 `diff` Performance Tradeoffs 33

6 Comparing Three Files 35
6.1 A Third Sample Input File 35
6.2 Detailed Description of `diff3` Normal Format 35
6.3 `diff3` Hunks 36
6.4 An Example of `diff3` Normal Format 37

7 Merging From a Common Ancestor 39
7.1 Selecting Which Changes to Incorporate 39
7.2 Marking Conflicts 40
7.3 Generating the Merged Output Directly 41
7.4 How `diff3` Merges Incomplete Lines 42
7.5 Saving the Changed File 42

8 Interactive Merging with `sdiff` 45
8.1 Specifying `diff` Options to `sdiff` 45
8.2 Merge Commands 45

9 Merging with `patch` 47
9.1 Selecting the `patch` Input Format 47
9.2 Applying Imperfect Patches 48
9.2.1 Applying Patches with Changed White Space .. 48
9.2.2 Applying Reversed Patches 48
9.2.3 Helping `patch` Find Inexact Matches 49
9.3 Removing Empty Files 50
9.4 Multiple Patches in a File 50
9.5 Messages and Questions from `patch` 51

10 Tips for Making Patch Distributions 53

11 Invoking `cmp` **55**
- 11.1 Options to `cmp` 55

12 Invoking `diff` **57**
- 12.1 Options to `diff` 57

13 Invoking `diff3` **65**
- 13.1 Options to `diff3` 65

14 Invoking `patch` **69**
- 14.1 Applying Patches in Other Directories 69
- 14.2 Backup File Names 70
- 14.3 Reject File Names 71
- 14.4 Options to `patch` 71

15 Invoking `sdiff` **77**
- 15.1 Options to `sdiff` 77

16 Incomplete Lines **81**

17 Future Projects **83**
- 17.1 Suggested Projects for Improving GNU `diff` and `patch` 83
  - 17.1.1 Handling Changes to the Directory Structure 83
  - 17.1.2 Files that are Neither Directories Nor Regular Files 83
  - 17.1.3 File Names that Contain Unusual Characters 84
  - 17.1.4 Arbitrary Limits 84
  - 17.1.5 Handling Files that Do Not Fit in Memory ... 84
  - 17.1.6 Ignoring Certain Changes 84
- 17.2 Reporting Bugs 85

Concept Index **87**

Overview

Computer users often find occasion to ask how two files differ. Perhaps one file is a newer version of the other file. Or maybe the two files started out as identical copies but were changed by different people.

You can use the `diff` command to show differences between two files, or each corresponding file in two directories. `diff` outputs differences between files line by line in any of several formats, selectable by command line options. This set of differences is often called a *diff* or *patch*. For files that are identical, `diff` normally produces no output; for binary (non-text) files, `diff` normally reports only that they are different.

You can use the `cmp` command to show the offsets and line numbers where two files differ. `cmp` can also show all the characters that differ between the two files, side by side. Another way to compare two files character by character is the Emacs command `M-x compare-windows`. See section "Other Window" in *The GNU Emacs Manual*, for more information on that command.

You can use the `diff3` command to show differences among three files. When two people have made independent changes to a common original, `diff3` can report the differences between the original and the two changed versions, and can produce a merged file that contains both persons' changes together with warnings about conflicts.

You can use the `sdiff` command to merge two files interactively.

You can use the set of differences produced by `diff` to distribute updates to text files (such as program source code) to other people. This method is especially useful when the differences are small compared to the complete files. Given `diff` output, you can use the `patch` program to update, or *patch*, a copy of the file. If you think of `diff` as subtracting one file from another to produce their difference, you can think of `patch` as adding the difference to one file to reproduce the other.

This manual first concentrates on making diffs, and later shows how to use diffs to update files.

GNU `diff` was written by Mike Haertel, David Hayes, Richard Stallman, Len Tower, and Paul Eggert. Wayne Davison designed and implemented the unified output format. The basic algorithm is described in "An O(ND) Difference Algorithm and its Variations", Eugene W. Myers, *Algorithmica* Vol. 1 No. 2, 1986, pp. 251–266; and in "A File Comparison Program", Webb Miller and Eugene W. Myers, *Software—Practice and Experience* Vol. 15 No. 11, 1985, pp. 1025–1040. The algorithm was independently discovered as described in "Algorithms for Approximate String Matching", E. Ukkonen, *Information and Control* Vol. 64, 1985, pp. 100–118.

GNU `diff3` was written by Randy Smith. GNU `sdiff` was written by Thomas Lord. GNU `cmp` was written by Torbjorn Granlund and David MacKenzie.

`patch` was written mainly by Larry Wall; the GNU enhancements were written mainly by Wayne Davison and David MacKenzie. Parts of this manual are adapted from a manual page written by Larry Wall, with his permission.

1 What Comparison Means

There are several ways to think about the differences between two files. One way to think of the differences is as a series of lines that were deleted from, inserted in, or changed in one file to produce the other file. `diff` compares two files line by line, finds groups of lines that differ, and reports each group of differing lines. It can report the differing lines in several formats, which have different purposes.

GNU `diff` can show whether files are different without detailing the differences. It also provides ways to suppress certain kinds of differences that are not important to you. Most commonly, such differences are changes in the amount of white space between words or lines. `diff` also provides ways to suppress differences in alphabetic case or in lines that match a regular expression that you provide. These options can accumulate; for example, you can ignore changes in both white space and alphabetic case.

Another way to think of the differences between two files is as a sequence of pairs of characters that can be either identical or different. `cmp` reports the differences between two files character by character, instead of line by line. As a result, it is more useful than `diff` for comparing binary files. For text files, `cmp` is useful mainly when you want to know only whether two files are identical.

To illustrate the effect that considering changes character by character can have compared with considering them line by line, think of what happens if a single newline character is added to the beginning of a file. If that file is then compared with an otherwise identical file that lacks the newline at the beginning, `diff` will report that a blank line has been added to the file, while `cmp` will report that almost every character of the two files differs.

`diff3` normally compares three input files line by line, finds groups of lines that differ, and reports each group of differing lines. Its output is designed to make it easy to inspect two different sets of changes to the same file.

1.1 Hunks

When comparing two files, `diff` finds sequences of lines common to both files, interspersed with groups of differing lines called *hunks*. Comparing two identical files yields one sequence of common lines and no hunks, because no lines differ. Comparing two entirely different files yields no common lines and one large hunk that contains all lines of both files. In general, there are many ways to match up lines between two given files. `diff` tries to minimize the total hunk size by finding large

sequences of common lines interspersed with small hunks of differing lines.

For example, suppose the file 'F' contains the three lines 'a', 'b', 'c', and the file 'G' contains the same three lines in reverse order 'c', 'b', 'a'. If diff finds the line 'c' as common, then the command 'diff F G' produces this output:

```
1,2d0
< a
< b
3a2,3
> b
> a
```

But if diff notices the common line 'b' instead, it produces this output:

```
1c1
< a
---
> c
3c3
< c
---
> a
```

It is also possible to find 'a' as the common line. diff does not always find an optimal matching between the files; it takes shortcuts to run faster. But its output is usually close to the shortest possible. You can adjust this tradeoff with the '--minimal' option (see Chapter 5 "diff Performance," page 33).

1.2 Suppressing Differences in Blank and Tab Spacing

The '-b' and '--ignore-space-change' options ignore white space at line end, and considers all other sequences of one or more white space characters to be equivalent. With these options, diff considers the following two lines to be equivalent, where '$' denotes the line end:

```
Here lyeth  muche rychnesse  in lytell space. -- John Heywood$
Here lyeth muche rychnesse in lytell space. -- John Heywood  $
```

The '-w' and '--ignore-all-space' options are stronger than '-b'. They ignore difference even if one file has white space where the other file has none. *White space* characters include tab, newline, vertical tab, form feed, carriage return, and space; some locales may define additiona 24 l characters to be white space. With these options, diff considers the following two lines to be equivalent, where '$' denotes the line end and '^M' denotes a carriage return:

```
Here lyeth  muche  rychnesse in lytell space.--  John Heywood$
He relyeth much erychnes seinly tells pace. --John Heywood ^M$
```

1.3 Suppressing Differences in Blank Lines

The '-B' and '--ignore-blank-lines' options ignore insertions or deletions of blank lines. These options normally affect only lines that are completely empty; they do not affect lines that look empty but contain space or tab characters. With these options, for example, a file containing

```
1.  A point is that which has no part.

2.  A line is breadthless length.
-- Euclid, The Elements, I
```

is considered identical to a file containing

```
1.  A point is that which has no part.
2.  A line is breadthless length.


-- Euclid, The Elements, I
```

1.4 Suppressing Case Differences

GNU `diff` can treat lowercase letters as equivalent to their uppercase counterparts, so that, for example, it considers 'Funky Stuff', 'funky STUFF', and 'fUNKy stuFf' to all be the same. To request this, use the '-i' or '--ignore-case' option.

1.5 Suppressing Lines Matching a Regular Expression

To ignore insertions and deletions of lines that match a regular expression, use the '-I *regexp*' or '--ignore-matching-lines=*regexp*' option. You should escape regular expressions that contain shell metacharacters to prevent the shell from expanding them. For example, 'diff -I '^[0-9]'' ignores all changes to lines beginning with a digit.

However, '-I' only ignores the insertion or deletion of lines that contain the regular expression if every changed line in the hunk—every insertion and every deletion—matches the regular expression. In other words, for each nonignorable change, `diff` prints the complete set of changes in its vicinity, including the ignorable ones.

You can specify more than one regular expression for lines to ignore by using more than one '-I' option. `diff` tries to match each line against each regular expression, starting with the last one given.

1.6 Summarizing Which Files Differ

When you only want to find out whether files are different, and you don't care what the differences are, you can use the summary output format. In this format, instead of showing the differences between the files, `diff` simply reports whether files differ. The '`-q`' and '`--brief`' options select this output format.

This format is especially useful when comparing the contents of two directories. It is also much faster than doing the normal line by line comparisons, because `diff` can stop analyzing the files as soon as it knows that there are any differences.

You can also get a brief indication of whether two files differ by using `cmp`. For files that are identical, `cmp` produces no output. When the files differ, by default, `cmp` outputs the byte offset and line number where the first difference occurs. You can use the '`-s`' option to suppress that information, so that `cmp` produces no output and reports whether the files differ using only its exit status (see Chapter 11 "Invoking cmp," page 55).

Unlike `diff`, `cmp` cannot compare directories; it can only compare two files.

1.7 Binary Files and Forcing Text Comparisons

If `diff` thinks that either of the two files it is comparing is binary (a non-text file), it normally treats that pair of files much as if the summary output format had been selected (see Section 1.6 "Brief," page 6), and reports only that the binary files are different. This is because line by line comparisons are usually not meaningful for binary files.

`diff` determines whether a file is text or binary by checking the first few bytes in the file; the exact number of bytes is system dependent, but it is typically several thousand. If every character in that part of the file is non-null, `diff` considers the file to be text; otherwise it considers the file to be binary.

Sometimes you might want to force `diff` to consider files to be text. For example, you might be comparing text files that contain null characters; `diff` would erroneously decide that those are non-text files. Or you might be comparing documents that are in a format used by a word processing system that uses null characters to indicate special formatting. You can force `diff` to consider all files to be text files, and compare them line by line, by using the '`-a`' or '`--text`' option. If the files you compare using this option do not in fact contain text, they will probably contain few newline characters, and the `diff` output will consist of

hunks showing differences between long lines of whatever characters the files contain.

You can also force `diff` to consider all files to be binary files, and report only whether they differ (but not how). Use the '`--brief`' option for this.

In operating systems that distinguish between text and binary files, `diff` normally reads and writes all data as text. Use the '`--binary`' option to force `diff` to read and write binary data instead. This option has no effect on a Posix-compliant system like GNU or traditional Unix. However, many personal computer operating systems represent the end of a line with a carriage return followed by a newline. On such systems, `diff` normally ignores these carriage returns on input and generates them at the end of each output line, but with the '`--binary`' option `diff` treats each carriage return as just another input character, and does not generate a carriage return at the end of each output line. This can be useful when dealing with non-text files that are meant to be interchanged with Posix-compliant systems.

If you want to compare two files byte by byte, you can use the `cmp` program with the '`-l`' option to show the values of each differing byte in the two files. With GNU `cmp`, you can also use the '`-c`' option to show the ASCII representation of those bytes. See Chapter 11 "Invoking cmp," page 55, for more information.

If `diff3` thinks that any of the files it is comparing is binary (a non-text file), it normally reports an error, because such comparisons are usually not useful. `diff3` uses the same test as `diff` to decide whether a file is binary. As with `diff`, if the input files contain a few non-text characters but otherwise are like text files, you can force `diff3` to consider all files to be text files and compare them line by line by using the '`-a`' or '`--text`' options.

2 diff Output Formats

diff has several mutually exclusive options for output format. The following sections describe each format, illustrating how diff reports the differences between two sample input files.

2.1 Two Sample Input Files

Here are two sample files that we will use in numerous examples to illustrate the output of diff and how various options can change it.

This is the file 'lao':

```
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
```

This is the file 'tzu':

```
The Nameless is the origin of Heaven and Earth;
The named is the mother of all things.

Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
```

In this example, the first hunk contains just the first two lines of 'lao', the second hunk contains the fourth line of 'lao' opposing the second and third lines of 'tzu', and the last hunk contains just the last three lines of 'tzu'.

2.2 Showing Differences Without Context

The "normal" diff output format shows each hunk of differences without any surrounding context. Sometimes such output is the clearest way to see how lines have changed, without the clutter of nearby

unchanged lines (although you can get similar results with the context or unified formats by using 0 lines of context). However, this format is no longer widely used for sending out patches; for that purpose, the context format (see Section 2.3.1 “Context Format,” page 11) and the unified format (see Section 2.3.2 “Unified Format,” page 14) are superior. Normal format is the default for compatibility with older versions of `diff` and the Posix standard.

2.2.1 Detailed Description of Normal Format

The normal output format consists of one or more hunks of differences; each hunk shows one area where the files differ. Normal format hunks look like this:

```
change-command
< from-file-line
< from-file-line. . .
---
> to-file-line
> to-file-line. . .
```

There are three types of change commands. Each consists of a line number or comma-separated range of lines in the first file, a single character indicating the kind of change to make, and a line number or comma-separated range of lines in the second file. All line numbers are the original line numbers in each file. The types of change commands are:

‘*l*a*r*’
: Add the lines in range *r* of the second file after line *l* of the first file. For example, ‘`8a12,15`’ means append lines 12–15 of file 2 after line 8 of file 1; or, if changing file 2 into file 1, delete lines 12–15 of file 2.

‘*f*c*t*’
: Replace the lines in range *f* of the first file with lines in range *t* of the second file. This is like a combined add and delete, but more compact. For example, ‘`5,7c8,10`’ means change lines 5–7 of file 1 to read as lines 8–10 of file 2; or, if changing file 2 into file 1, change lines 8–10 of file 2 to read as lines 5–7 of file 1.

‘*r*d*l*’
: Delete the lines in range *r* from the first file; line *l* is where they would have appeared in the second file had they not been deleted. For example, ‘`5,7d3`’ means delete lines 5–7 of file 1; or, if changing file 2 into file 1, append lines 5–7 of file 1 after line 3 of file 2.

2.2.2 An Example of Normal Format

Here is the output of the command 'diff lao tzu' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files). Notice that it shows only the lines that are different between the two files.

```
1,2d0
< The Way that can be told of is not the eternal Way;
< The name that can be named is not the eternal name.
4c2,3
< The Named is the mother of all things.
---
> The named is the mother of all things.
>
11a11,13
> They both may be called deep and profound.
> Deeper and more profound,
> The door of all subtleties!
```

2.3 Showing Differences in Their Context

Usually, when you are looking at the differences between files, you will also want to see the parts of the files near the lines that differ, to help you understand exactly what has changed. These nearby parts of the files are called the *context*.

GNU diff provides two output formats that show context around the differing lines: *context format* and *unified format*. It can optionally show in which function or section of the file the differing lines are found.

If you are distributing new versions of files to other people in the form of diff output, you should use one of the output formats that show context so that they can apply the diffs even if they have made small changes of their own to the files. patch can apply the diffs in this case by searching in the files for the lines of context around the differing lines; if those lines are actually a few lines away from where the diff says they are, patch can adjust the line numbers accordingly and still apply the diff correctly. See Section 9.2 "Imperfect," page 48, for more information on using patch to apply imperfect diffs.

2.3.1 Context Format

The context output format shows several lines of context around the lines that differ. It is the standard format for distributing updates to source code.

To select this output format, use the '-C *lines*', '--context[=*lines*]', or '-c' option. The argument *lines* that some of these options take is

the number of lines of context to show. If you do not specify *lines*, it defaults to three. For proper operation, `patch` typically needs at least two lines of context.

2.3.1.1 Detailed Description of Context Format

The context output format starts with a two-line header, which looks like this:

```
*** from-file from-file-modification-time
--- to-file to-file-modification time
```

You can change the header's content with the '`-L` *label*' or '`--label=`*label*' option; see Section 2.3.4 "Alternate Names," page 16.

Next come one or more hunks of differences; each hunk shows one area where the files differ. Context format hunks look like this:

```
***************
*** from-file-line-range ****
  from-file-line
  from-file-line...
--- to-file-line-range ----
  to-file-line
  to-file-line...
```

The lines of context around the lines that differ start with two space characters. The lines that differ between the two files start with one of the following indicator characters, followed by a space character:

'`!`' A line that is part of a group of one or more lines that changed between the two files. There is a corresponding group of lines marked with '`!`' in the part of this hunk for the other file.

'`+`' An "inserted" line in the second file that corresponds to nothing in the first file.

'`-`' A "deleted" line in the first file that corresponds to nothing in the second file.

If all of the changes in a hunk are insertions, the lines of *from-file* are omitted. If all of the changes are deletions, the lines of *to-file* are omitted.

2.3.1.2 An Example of Context Format

Here is the output of '`diff -c lao tzu`' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files). Notice that up to three lines that are not different are shown around each line that is different; they are the context lines. Also notice that the first two hunks have run together, because their contents overlap.

```
*** lao Sat Jan 26 23:30:39 1991
--- tzu Sat Jan 26 23:30:50 1991
***************
*** 1,7 ****
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
! The Named is the mother of all things.
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
--- 1,6 ----
  The Nameless is the origin of Heaven and Earth;
! The named is the mother of all things.
!
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
***************
*** 9,11 ****
--- 8,13 ----
  The two are the same,
  But after they are produced,
    they have different names.
+ They both may be called deep and profound.
+ Deeper and more profound,
+ The door of all subtleties!
```

2.3.1.3 An Example of Context Format with Less Context

Here is the output of 'diff --context=1 lao tzu' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files). Notice that at most one context line is reported here.

```
*** lao Sat Jan 26 23:30:39 1991
--- tzu Sat Jan 26 23:30:50 1991
***************
*** 1,5 ****
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
! The Named is the mother of all things.
  Therefore let there always be non-being,
--- 1,4 ----
  The Nameless is the origin of Heaven and Earth;
! The named is the mother of all things.
!
  Therefore let there always be non-being,
***************
*** 11 ****
--- 10,13 ----
    they have different names.
+ They both may be called deep and profound.
```

```
+ Deeper and more profound,
+ The door of all subtleties!
```

2.3.2 Unified Format

The unified output format is a variation on the context format that is more compact because it omits redundant context lines. To select this output format, use the '-U *lines*', '--unified[=*lines*]', or '-u' option. The argument *lines* is the number of lines of context to show. When it is not given, it defaults to three.

At present, only GNU diff can produce this format and only GNU patch can automatically apply diffs in this format. For proper operation, patch typically needs at least two lines of context.

2.3.2.1 Detailed Description of Unified Format

The unified output format starts with a two-line header, which looks like this:

```
--- from-file from-file-modification-time
+++ to-file to-file-modification-time
```

You can change the header's content with the '-L *label*' or '--label=*label*' option; see See Section 2.3.4 "Alternate Names," page 16.

Next come one or more hunks of differences; each hunk shows one area where the files differ. Unified format hunks look like this:

```
@@ from-file-range to-file-range @@
 line-from-either-file
 line-from-either-file...
```

The lines common to both files begin with a space character. The lines that actually differ between the two files have one of the following indicator characters in the left column:

'+' A line was added here to the first file.

'-' A line was removed here from the first file.

2.3.2.2 An Example of Unified Format

Here is the output of the command 'diff -u lao tzu' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files):

```
--- lao Sat Jan 26 23:30:39 1991
+++ tzu Sat Jan 26 23:30:50 1991
@@ -1,7 +1,6 @@
-The Way that can be told of is not the eternal Way;
```

```
-The name that can be named is not the eternal name.
 The Nameless is the origin of Heaven and Earth;
-The Named is the mother of all things.
+The named is the mother of all things.
+
 Therefore let there always be non-being,
   so we may see their subtlety,
 And let there always be being,
@@ -9,3 +8,6 @@
 The two are the same,
 But after they are produced,
   they have different names.
+They both may be called deep and profound.
+Deeper and more profound,
+The door of all subtleties!
```

2.3.3 Showing Which Sections Differences Are in

Sometimes you might want to know which part of the files each change falls in. If the files are source code, this could mean which function was changed. If the files are documents, it could mean which chapter or appendix was changed. GNU `diff` can show this by displaying the nearest section heading line that precedes the differing lines. Which lines are "section headings" is determined by a regular expression.

2.3.3.1 Showing Lines That Match Regular Expressions

To show in which sections differences occur for files that are not source code for C or similar languages, use the '`-F` *regexp*' or '`--show-function-line=`*regexp*' option. `diff` considers lines that match the argument *regexp* to be the beginning of a section of the file. Here are suggested regular expressions for some common languages:

'`^[A-Za-z_]`'
: C, C++, Prolog

'`^(`'
: Lisp

'`^@\(chapter\|appendix\|unnumbered\|chapheading\)`'
: Texinfo

This option does not automatically select an output format; in order to use it, you must select the context format (see Section 2.3.1 "Context Format," page 11) or unified format (see Section 2.3.2 "Unified Format," page 14). In other output formats it has no effect.

The '`-F`' and '`--show-function-line`' options find the nearest unchanged line that precedes each hunk of differences and matches the given regular expression. Then they add that line to the end of the line

of asterisks in the context format, or to the '@@' line in unified format. If no matching line exists, they leave the output for that hunk unchanged. If that line is more than 40 characters long, they output only the first 40 characters. You can specify more than one regular expression for such lines; `diff` tries to match each line against each regular expression, starting with the last one given. This means that you can use '-p' and '-F' together, if you wish.

2.3.3.2 Showing C Function Headings

To show in which functions differences occur for C and similar languages, you can use the '-p' or '--show-c-function' option. This option automatically defaults to the context output format (see Section 2.3.1 "Context Format," page 11), with the default number of lines of context. You can override that number with '-C *lines*' elsewhere in the command line. You can override both the format and the number with '-U *lines*' elsewhere in the command line.

The '-p' and '--show-c-function' options are equivalent to '-F'^[_a-zA-Z$]'' if the unified format is specified, otherwise '-c -F'^[_a-zA-Z$]'' (see Section 2.3.3.1 "Specified Headings," page 15). GNU `diff` provides them for the sake of convenience.

2.3.4 Showing Alternate File Names

If you are comparing two files that have meaningless or uninformative names, you might want `diff` to show alternate names in the header of the context and unified output formats. To do this, use the '-L *label*' or '--label=*label*' option. The first time you give this option, its argument replaces the name and date of the first file in the header; the second time, its argument replaces the name and date of the second file. If you give this option more than twice, `diff` reports an error. The '-L' option does not affect the file names in the `pr` header when the '-l' or '--paginate' option is used (see Section 4.2 "Pagination," page 31).

Here are the first two lines of the output from 'diff -C2 -Loriginal -Lmodified lao tzu':

```
*** original
--- modified
```

2.4 Showing Differences Side by Side

`diff` can produce a side by side difference listing of two files. The files are listed in two columns with a gutter between them. The gutter contains one of the following markers:

white space
: The corresponding lines are in common. That is, either the lines are identical, or the difference is ignored because of one of the '--ignore' options (see Section 1.2 "White Space," page 4).

'|' The corresponding lines differ, and they are either both complete or both incomplete.

'<' The files differ and only the first file contains the line.

'>' The files differ and only the second file contains the line.

'(' Only the first file contains the line, but the difference is ignored.

')' Only the second file contains the line, but the difference is ignored.

'\' The corresponding lines differ, and only the first line is incomplete.

'/' The corresponding lines differ, and only the second line is incomplete.

Normally, an output line is incomplete if and only if the lines that it contains are incomplete; See Chapter 16 "Incomplete Lines," page 81. However, when an output line represents two differing lines, one might be incomplete while the other is not. In this case, the output line is complete, but its the gutter is marked '\' if the first line is incomplete, '/' if the second line is.

Side by side format is sometimes easiest to read, but it has limitations. It generates much wider output than usual, and truncates lines that are too long to fit. Also, it relies on lining up output more heavily than usual, so its output looks particularly bad if you use varying width fonts, nonstandard tab stops, or nonprinting characters.

You can use the `sdiff` command to interactively merge side by side differences. See Chapter 8 "Interactive Merging," page 45, for more information on merging files.

2.5 Controlling Side by Side Format

The '-y' or '--side-by-side' option selects side by side format. Because side by side output lines contain two input lines, they are wider than usual. They are normally 130 columns, which can fit onto a traditional printer line. You can set the length of output lines with the '-W *columns*' or '--width=*columns*' option. The output line is split into two halves of equal length, separated by a small gutter to mark differences;

the right half is aligned to a tab stop so that tabs line up. Input lines that are too long to fit in half of an output line are truncated for output.

The '--left-column' option prints only the left column of two common lines. The '--suppress-common-lines' option suppresses common lines entirely.

2.5.1 An Example of Side by Side Format

Here is the output of the command 'diff -y -W 72 lao tzu' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files).

```
The Way that can be told of is <
The name that can be named is  <
The Nameless is the origin of    The Nameless is the origin of
The Named is the mother of all | The named is the mother of all
                               >
Therefore let there always be    Therefore let there always be
  so we may see their subtlet      so we may see their subtlet
And let there always be being    And let there always be being
  so we may see their outcome      so we may see their outcome
The two are the same,            The two are the same,
But after they are produced,     But after they are produced,
  they have different names.       they have different names.
                               > They both may be called deep
                               > Deeper and more profound,
                               > The door of all subtleties!
```

2.6 Making Edit Scripts

Several output modes produce command scripts for editing *from-file* to produce *to-file*.

2.6.1 ed Scripts

diff can produce commands that direct the ed text editor to change the first file into the second file. Long ago, this was the only output mode that was suitable for editing one file into another automatically; today, with patch, it is almost obsolete. Use the '-e' or '--ed' option to select this output format.

Like the normal format (see Section 2.2 "Normal," page 9), this output format does not show any context; unlike the normal format, it does not include the information necessary to apply the diff in reverse (to produce the first file if all you have is the second file and the diff).

If the file 'd' contains the output of 'diff -e old new', then the command '(cat d && echo w) | ed - old' edits 'old' to make it a copy of 'new'.

More generally, if 'd1', 'd2', ..., 'dN' contain the outputs of 'diff -e old new1', 'diff -e new1 new2', ..., 'diff -e newN-1 newN', respectively, then the command '(cat d1 d2 . . . dN && echo w) | ed - old' edits 'old' to make it a copy of 'newN'.

2.6.1.1 Detailed Description of ed Format

The ed output format consists of one or more hunks of differences. The changes closest to the ends of the files come first so that commands that change the number of lines do not affect how ed interprets line numbers in succeeding commands. ed format hunks look like this:

```
change-command
to-file-line
to-file-line...
.
```

Because ed uses a single period on a line to indicate the end of input, GNU diff protects lines of changes that contain a single period on a line by writing two periods instead, then writing a subsequent ed command to change the two periods into one. The ed format cannot represent an incomplete line, so if the second file ends in a changed incomplete line, diff reports an error and then pretends that a newline was appended.

There are three types of change commands. Each consists of a line number or comma-separated range of lines in the first file and a single character indicating the kind of change to make. All line numbers are the original line numbers in the file. The types of change commands are:

'*l*a' Add text from the second file after line *l* in the first file. For example, '8a' means to add the following lines after line 8 of file 1.

'*r*c' Replace the lines in range *r* in the first file with the following lines. Like a combined add and delete, but more compact. For example, '5,7c' means change lines 5–7 of file 1 to read as the text file 2.

'*r*d' Delete the lines in range *r* from the first file. For example, '5,7d' means delete lines 5–7 of file 1.

2.6.1.2 Example ed Script

Here is the output of 'diff -e lao tzu' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files):

```
11a
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
.
4c
The named is the mother of all things.

.
1,2d
```

2.6.2 Forward `ed` Scripts

`diff` can produce output that is like an `ed` script, but with hunks in forward (front to back) order. The format of the commands is also changed slightly: command characters precede the lines they modify, spaces separate line numbers in ranges, and no attempt is made to disambiguate hunk lines consisting of a single period. Like `ed` format, forward `ed` format cannot represent incomplete lines.

Forward `ed` format is not very useful, because neither `ed` nor `patch` can apply diffs in this format. It exists mainly for compatibility with older versions of `diff`. Use the '`-f`' or '`--forward-ed`' option to select it.

2.6.3 RCS Scripts

The RCS output format is designed specifically for use by the Revision Control System, which is a set of free programs used for organizing different versions and systems of files. Use the '`-n`' or '`--rcs`' option to select this output format. It is like the forward `ed` format (see Section 2.6.2 "Forward ed," page 20), but it can represent arbitrary changes to the contents of a file because it avoids the forward `ed` format's problems with lines consisting of a single period and with incomplete lines. Instead of ending text sections with a line consisting of a single period, each command specifies the number of lines it affects; a combination of the '`a`' and '`d`' commands are used instead of '`c`'. Also, if the second file ends in a changed incomplete line, then the output also ends in an incomplete line.

Here is the output of '`diff -n lao tzu`' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files):

```
d1 2
d4 1
a4 2
The named is the mother of all things.
```

```
a11 3
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
```

2.7 Merging Files with If-then-else

You can use `diff` to merge two files of C source code. The output of `diff` in this format contains all the lines of both files. Lines common to both files are output just once; the differing parts are separated by the C preprocessor directives `#ifdef` *name* or `#ifndef` *name*, `#else`, and `#endif`. When compiling the output, you select which version to use by either defining or leaving undefined the macro *name*.

To merge two files, use `diff` with the '`-D` *name*' or '`--ifdef=`*name*' option. The argument *name* is the C preprocessor identifier to use in the `#ifdef` and `#ifndef` directives.

For example, if you change an instance of `wait (&s)` to `waitpid (-1, &s, 0)` and then merge the old and new files with the '`--ifdef=HAVE_WAITPID`' option, then the affected part of your code might look like this:

```
    do {
#ifndef HAVE_WAITPID
        if ((w = wait (&s)) < 0  &&  errno != EINTR)
#else /* HAVE_WAITPID */
        if ((w = waitpid (-1, &s, 0)) < 0  &&  errno != EINTR)
#endif /* HAVE_WAITPID */
            return w;
    } while (w != child);
```

You can specify formats for languages other than C by using line group formats and line formats, as described in the next sections.

2.7.1 Line Group Formats

Line group formats let you specify formats suitable for many applications that allow if-then-else input, including programming languages and text formatting languages. A line group format specifies the output format for a contiguous group of similar lines.

For example, the following command compares the TeX files '`old`' and '`new`', and outputs a merged file in which old regions are surrounded by '`\begin{em}`'-'`\end{em}`' lines, and new regions are surrounded by '`\begin{bf}`'-'`\end{bf}`' lines.

```
diff \
   --old-group-format='\begin{em}
%<\end{em}
```

```
    ' \
       --new-group-format='\begin{bf}
    %>\end{bf}
    ' \
       old new
```

The following command is equivalent to the above example, but it is a little more verbose, because it spells out the default line group formats.

```
    diff \
       --old-group-format='\begin{em}
    %<\end{em}
    ' \
       --new-group-format='\begin{bf}
    %>\end{bf}
    ' \
       --unchanged-group-format='%=' \
       --changed-group-format='\begin{em}
    %<\end{em}
    \begin{bf}
    %>\end{bf}
    ' \
       old new
```

Here is a more advanced example, which outputs a diff listing with headers containing line numbers in a “plain English” style.

```
diff \
 --unchanged-group-format='' \
 --old-group-format='-------- %dn line%(n=1?:s) deleted at %df:
%<' \
 --new-group-format='-------- %dN line%(N=1?:s) added after %de:
%>' \
 --changed-group-format='-------- %dn line%(n=1?:s) changed at %df:
%<-------- to:
%>' \
 old new
```

To specify a line group format, use `diff` with one of the options listed below. You can specify up to four line group formats, one for each kind of line group. You should quote *format*, because it typically contains shell metacharacters.

‘`--old-group-format=`*format*’
: These line groups are hunks containing only lines from the first file. The default old group format is the same as the changed group format if it is specified; otherwise it is a format that outputs the line group as-is.

‘`--new-group-format=`*format*’
: These line groups are hunks containing only lines from the second file. The default new group format is same as the the changed group format if it is specified; otherwise it is a format that outputs the line group as-is.

‘--changed-group-format=*format*’
: These line groups are hunks containing lines from both files. The default changed group format is the concatenation of the old and new group formats.

‘--unchanged-group-format=*format*’
: These line groups contain lines common to both files. The default unchanged group format is a format that outputs the line group as-is.

In a line group format, ordinary characters represent themselves; conversion specifications start with ‘%’ and have one of the following forms.

‘%<’
: stands for the lines from the first file, including the trailing newline. Each line is formatted according to the old line format (see Section 2.7.2 “Line Formats,” page 24).

‘%>’
: stands for the lines from the second file, including the trailing newline. Each line is formatted according to the new line format.

‘%=’
: stands for the lines common to both files, including the trailing newline. Each line is formatted according to the unchanged line format.

‘%%’
: stands for ‘%’.

‘%c'*C*'’
: where *C* is a single character, stands for *C*. *C* may not be a backslash or an apostrophe. For example, ‘%c':'’ stands for a colon, even inside the then-part of an if-then-else format, which a colon would normally terminate.

‘%c'*O*'’
: where *O* is a string of 1, 2, or 3 octal digits, stands for the character with octal code *O*. For example, ‘%c'\0'’ stands for a null character.

‘*Fn*’
: where *F* is a printf conversion specification and *n* is one of the following letters, stands for *n*’s value formatted with *F*.

  ‘e’
  : The line number of the line just before the group in the old file.

  ‘f’
  : The line number of the first line in the group in the old file; equals *e* + 1.

  ‘l’
  : The line number of the last line in the group in the old file.

  ‘m’
  : The line number of the line just after the group in the old file; equals *l* + 1.

'n' The number of lines in the group in the old file; equals `l` - `f` + 1.

'E, F, L, M, N'
Likewise, for lines in the new file.

The `printf` conversion specification can be '%d', '%o', '%x', or '%X', specifying decimal, octal, lower case hexadecimal, or upper case hexadecimal output respectively. After the '%' the following options can appear in sequence: a '-' specifying left-justification; an integer specifying the minimum field width; and a period followed by an optional integer specifying the minimum number of digits. For example, '%5dN' prints the number of new lines in the group in a field of width 5 characters, using the `printf` format `"%5d"`.

'(*A*=*B*?*T*:*E*)'
If *A* equals *B* then *T* else *E*. *A* and *B* are each either a decimal constant or a single letter interpreted as above. This format spec is equivalent to *T* if *A*'s value equals *B*'s; otherwise it is equivalent to *E*.

For example, '%(N=0?no:%dN) line%(N=1?:s)' is equivalent to 'no lines' if *N* (the number of lines in the group in the the new file) is 0, to '1 line' if *N* is 1, and to '%dN lines' otherwise.

2.7.2 Line Formats

Line formats control how each line taken from an input file is output as part of a line group in if-then-else format.

For example, the following command outputs text with a one-column change indicator to the left of the text. The first column of output is '-' for deleted lines, '|' for added lines, and a space for unchanged lines. The formats contain newline characters where newlines are desired on output.

```
diff \
   --old-line-format='-%l
' \
   --new-line-format='|%l
' \
   --unchanged-line-format=' %l
' \
   old new
```

To specify a line format, use one of the following options. You should quote *format*, since it often contains shell metacharacters.

'--old-line-format=*format*'
formats lines just from the first file.

'--new-line-format=*format*'
: formats lines just from the second file.

'--unchanged-line-format=*format*'
: formats lines common to both files.

'--line-format=*format*'
: formats all lines; in effect, it sets all three above options simultaneously.

In a line format, ordinary characters represent themselves; conversion specifications start with '%' and have one of the following forms.

'%l'
: stands for the the contents of the line, not counting its trailing newline (if any). This format ignores whether the line is incomplete; See Chapter 16 "Incomplete Lines," page 81.

'%L'
: stands for the the contents of the line, including its trailing newline (if any). If a line is incomplete, this format preserves its incompleteness.

'%%'
: stands for '%'.

'%c'*C*''
: where *C* is a single character, stands for *C*. *C* may not be a backslash or an apostrophe. For example, '%c':'' stands for a colon.

'%c'*O*''
: where *O* is a string of 1, 2, or 3 octal digits, stands for the character with octal code *O*. For example, '%c'\0'' stands for a null character.

'*F*n'
: where *F* is a printf conversion specification, stands for the line number formatted with *F*. For example, '%.5dn' prints the line number using the printf format "%.5d". See Section 2.7.1 "Line Group Formats," page 21, for more about printf conversion specifications.

The default line format is '%l' followed by a newline character.

If the input contains tab characters and it is important that they line up on output, you should ensure that '%l' or '%L' in a line format is just after a tab stop (e.g. by preceding '%l' or '%L' with a tab character), or you should use the '-t' or '--expand-tabs' option.

Taken together, the line and line group formats let you specify many different formats. For example, the following command uses a format similar to diff's normal format. You can tailor this command to get fine control over diff's output.

```
diff \
   --old-line-format='< %l
' \
   --new-line-format='> %l
```

```
' \
   --old-group-format='%df%(f=l?:,%dl)d%dE
%<' \
   --new-group-format='%dea%dF%(F=L?:,%dL)
%>' \
   --changed-group-format='%df%(f=l?:,%dl)c%dF%(F=L?:,%dL)
%<---
%>' \
   --unchanged-group-format='' \
   old new
```

2.7.3 Detailed Description of If-then-else Format

For lines common to both files, `diff` uses the unchanged line group format. For each hunk of differences in the merged output format, if the hunk contains only lines from the first file, `diff` uses the old line group format; if the hunk contains only lines from the second file, `diff` uses the new group format; otherwise, `diff` uses the changed group format.

The old, new, and unchanged line formats specify the output format of lines from the first file, lines from the second file, and lines common to both files, respectively.

The option '`--ifdef=`*name*' is equivalent to the following sequence of options using shell syntax:

```
--old-group-format='#ifndef name
%<#endif /* not name */
' \
--new-group-format='#ifdef name
%>#endif /* name */
' \
--unchanged-group-format='%=' \
--changed-group-format='#ifndef name
%<#else /* name */
%>#endif /* name */
'
```

You should carefully check the `diff` output for proper nesting. For example, when using the the '`-D` *name*' or '`--ifdef=`*name*' option, you should check that if the differing lines contain any of the C preprocessor directives '`#ifdef`', '`#ifndef`', '`#else`', '`#elif`', or '`#endif`', they are nested properly and match. If they don't, you must make corrections manually. It is a good idea to carefully check the resulting code anyway to make sure that it really does what you want it to; depending on how the input files were produced, the output might contain duplicate or otherwise incorrect code.

The `patch` '`-D` *name*' option behaves just like the `diff` '`-D` *name*' option, except it operates on a file and a diff to produce a merged file; See Section 14.4 "patch Options," page 71.

2.7.4 An Example of If-then-else Format

Here is the output of '`diff -DTWO lao tzu`' (see Section 2.1 "Sample diff Input," page 9, for the complete contents of the two files):

```
#ifndef TWO
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
#endif /* not TWO */
The Nameless is the origin of Heaven and Earth;
#ifndef TWO
The Named is the mother of all things.
#else /* TWO */
The named is the mother of all things.

#endif /* TWO */
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
#ifdef TWO
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
#endif /* TWO */
```

3 Comparing Directories

You can use `diff` to compare some or all of the files in two directory trees. When both file name arguments to `diff` are directories, it compares each file that is contained in both directories, examining file names in alphabetical order. Normally `diff` is silent about pairs of files that contain no differences, but if you use the '`-s`' or '`--report-identical-files`' option, it reports pairs of identical files. Normally `diff` reports subdirectories common to both directories without comparing subdirectories' files, but if you use the '`-r`' or '`--recursive`' option, it compares every corresponding pair of files in the directory trees, as many levels deep as they go.

For file names that are in only one of the directories, `diff` normally does not show the contents of the file that exists; it reports only that the file exists in that directory and not in the other. You can make `diff` act as though the file existed but was empty in the other directory, so that it outputs the entire contents of the file that actually exists. (It is output as either an insertion or a deletion, depending on whether it is in the first or the second directory given.) To do this, use the '`-N`' or '`--new-file`' option.

If the older directory contains one or more large files that are not in the newer directory, you can make the patch smaller by using the '`-P`' or '`--unidirectional-new-file`' option instead of '`-N`'. This option is like '`-N`' except that it only inserts the contents of files that appear in the second directory but not the first (that is, files that were added). At the top of the patch, write instructions for the user applying the patch to remove the files that were deleted before applying the patch. See Chapter 10 "Making Patches," page 53, for more discussion of making patches for distribution.

To ignore some files while comparing directories, use the '`-x` *`pattern`*' or '`--exclude=`*`pattern`*' option. This option ignores any files or subdirectories whose base names match the shell pattern *`pattern`*. Unlike in the shell, a period at the start of the base of a file name matches a wildcard at the start of a pattern. You should enclose *`pattern`* in quotes so that the shell does not expand it. For example, the option '`-x '*.[ao]'`' ignores any file whose name ends with '`.a`' or '`.o`'.

This option accumulates if you specify it more than once. For example, using the options '`-x 'RCS' -x '*,v'`' ignores any file or subdirectory whose base name is '`RCS`' or ends with '`,v`'.

If you need to give this option many times, you can instead put the patterns in a file, one pattern per line, and use the '`-X` *`file`*' or '`--exclude-from=`*`file`*' option.

If you have been comparing two directories and stopped partway through, later you might want to continue where you left off. You can do this by using the '`-S` *`file`*' or '`--starting-file=`*`file`*' option. This compares only the file *`file`* and all alphabetically later files in the topmost directory level.

4 Making `diff` Output Prettier

`diff` provides several ways to adjust the appearance of its output. These adjustments can be applied to any output format.

4.1 Preserving Tabstop Alignment

The lines of text in some of the `diff` output formats are preceded by one or two characters that indicate whether the text is inserted, deleted, or changed. The addition of those characters can cause tabs to move to the next tabstop, throwing off the alignment of columns in the line. GNU `diff` provides two ways to make tab-aligned columns line up correctly.

The first way is to have `diff` convert all tabs into the correct number of spaces before outputting them; select this method with the '`-t`' or '`--expand-tabs`' option. `diff` assumes that tabstops are set every 8 columns. To use this form of output with `patch`, you must give `patch` the '`-l`' or '`--ignore-white-space`' option (see Section 9.2.1 "Changed White Space," page 48, for more information).

The other method for making tabs line up correctly is to add a tab character instead of a space after the indicator character at the beginning of the line. This ensures that all following tab characters are in the same position relative to tabstops that they were in the original files, so that the output is aligned correctly. Its disadvantage is that it can make long lines too long to fit on one line of the screen or the paper. It also does not work with the unified output format, which does not have a space character after the change type indicator character. Select this method with the '`-T`' or '`--initial-tab`' option.

4.2 Paginating `diff` Output

It can be convenient to have long output page-numbered and time-stamped. The '`-l`' and '`--paginate`' options do this by sending the `diff` output through the `pr` program. Here is what the page header might look like for '`diff -lc lao tzu`':

```
Mar 11 13:37 1991  diff -lc lao tzu Page 1
```

5 `diff` Performance Tradeoffs

GNU `diff` runs quite efficiently; however, in some circumstances you can cause it to run faster or produce a more compact set of changes. There are two ways that you can affect the performance of GNU `diff` by changing the way it compares files.

Performance has more than one dimension. These options improve one aspect of performance at the cost of another, or they improve performance in some cases while hurting it in others.

The way that GNU `diff` determines which lines have changed always comes up with a near-minimal set of differences. Usually it is good enough for practical purposes. If the `diff` output is large, you might want `diff` to use a modified algorithm that sometimes produces a smaller set of differences. The '`-d`' or '`--minimal`' option does this; however, it can also cause `diff` to run more slowly than usual, so it is not the default behavior.

When the files you are comparing are large and have small groups of changes scattered throughout them, you can use the '`-H`' or '`--speed-large-files`' option to make a different modification to the algorithm that `diff` uses. If the input files have a constant small density of changes, this option speeds up the comparisons without changing the output. If not, `diff` might produce a larger set of differences; however, the output will still be correct.

Normally `diff` discards the prefix and suffix that is common to both files before it attempts to find a minimal set of differences. This makes `diff` run faster, but occasionally it may produce non-minimal output. The '`--horizon-lines=`*`lines`*' option prevents `diff` from discarding the last *`lines`* lines of the prefix and the first *`lines`* lines of the suffix. This gives `diff` further opportunities to find a minimal output.

6 Comparing Three Files

Use the program `diff3` to compare three files and show any differences among them. (`diff3` can also merge files; see Chapter 7 "diff3 Merging," page 39).

The "normal" `diff3` output format shows each hunk of differences without surrounding context. Hunks are labeled depending on whether they are two-way or three-way, and lines are annotated by their location in the input files.

See Chapter 13 "Invoking diff3," page 65, for more information on how to run `diff3`.

6.1 A Third Sample Input File

Here is a third sample file that will be used in examples to illustrate the output of `diff3` and how various options can change it. The first two files are the same that we used for `diff` (see Section 2.1 "Sample diff Input," page 9). This is the third sample file, called '`tao`':

```
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The named is the mother of all things.

Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their result.
The two are the same,
But after they are produced,
  they have different names.

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
```

6.2 Detailed Description of `diff3` Normal Format

Each hunk begins with a line marked '`====`'. Three-way hunks have plain '`====`' lines, and two-way hunks have '`1`', '`2`', or '`3`' appended to specify which of the three input files differ in that hunk. The hunks contain copies of two or three sets of input lines each preceded by one or two commands identifying where the lines came from.

Normally, two spaces precede each copy of an input line to distinguish it from the commands. But with the '`-T`' or '`--initial-tab`' option, `diff3` uses a tab instead of two spaces; this lines up tabs correctly. See Section 4.1 "Tabs," page 31, for more information.

Commands take the following forms:

'*file*:*l*a' This hunk appears after line *l* of file *file*, and contains no lines in that file. To edit this file to yield the other files, one must append hunk lines taken from the other files. For example, '1:11a' means that the hunk follows line 11 in the first file and contains no lines from that file.

'*file*:*r*c' This hunk contains the lines in the range *r* of file *file*. The range *r* is a comma-separated pair of line numbers, or just one number if the range is a singleton. To edit this file to yield the other files, one must change the specified lines to be the lines taken from the other files. For example, '2:11,13c' means that the hunk contains lines 11 through 13 from the second file.

If the last line in a set of input lines is incomplete (see Chapter 16 "Incomplete Lines," page 81), it is distinguished on output from a full line by a following line that starts with '\'.

6.3 diff3 Hunks

Groups of lines that differ in two or three of the input files are called *diff3 hunks*, by analogy with diff hunks (see Section 1.1 "Hunks," page 3). If all three input files differ in a diff3 hunk, the hunk is called a *three-way hunk*; if just two input files differ, it is a *two-way hunk*.

As with diff, several solutions are possible. When comparing the files 'A', 'B', and 'C', diff3 normally finds diff3 hunks by merging the two-way hunks output by the two commands 'diff A B' and 'diff A C'. This does not necessarily minimize the size of the output, but exceptions should be rare.

For example, suppose 'F' contains the three lines 'a', 'b', 'f', 'G' contains the lines 'g', 'b', 'g', and 'H' contains the lines 'a', 'b', 'h'. 'diff3 F G H' might output the following:

```
====2
1:1c
3:1c
  a
2:1c
  g
====
1:3c
  f
2:3c
  g
3:3c
  h
```

because it found a two-way hunk containing 'a' in the first and third files and 'g' in the second file, then the single line 'b' common to all three files, then a three-way hunk containing the last line of each file.

6.4 An Example of diff3 Normal Format

Here is the output of the command 'diff3 lao tzu tao' (see Section 6.1 "Sample diff3 Input," page 35, for the complete contents of the files). Notice that it shows only the lines that are different among the three files.

```
====2
1:1,2c
3:1,2c
  The Way that can be told of is not the eternal Way;
  The name that can be named is not the eternal name.
2:0a
====1
1:4c
  The Named is the mother of all things.
2:2,3c
3:4,5c
  The named is the mother of all things.

====3
1:8c
2:7c
    so we may see their outcome.
3:9c
    so we may see their result.
====
1:11a
2:11,13c
  They both may be called deep and profound.
  Deeper and more profound,
  The door of all subtleties!
3:13,14c

    -- The Way of Lao-Tzu, tr. Wing-tsit Chan
```

7 Merging From a Common Ancestor

When two people have made changes to copies of the same file, `diff3` can produce a merged output that contains both sets of changes together with warnings about conflicts.

One might imagine programs with names like `diff4` and `diff5` to compare more than three files simultaneously, but in practice the need rarely arises. You can use `diff3` to merge three or more sets of changes to a file by merging two change sets at a time.

`diff3` can incorporate changes from two modified versions into a common preceding version. This lets you merge the sets of changes represented by the two newer files. Specify the common ancestor version as the second argument and the two newer versions as the first and third arguments, like this:

```
diff3 mine older yours
```

You can remember the order of the arguments by noting that they are in alphabetical order.

You can think of this as subtracting *older* from *yours* and adding the result to *mine*, or as merging into *mine* the changes that would turn *older* into *yours*. This merging is well-defined as long as *mine* and *older* match in the neighborhood of each such change. This fails to be true when all three input files differ or when only *older* differs; we call this a *conflict*. When all three input files differ, we call the conflict an *overlap*.

`diff3` gives you several ways to handle overlaps and conflicts. You can omit overlaps or conflicts, or select only overlaps, or mark conflicts with special '`<<<<<<<`' and '`>>>>>>>`' lines.

`diff3` can output the merge results as an `ed` script that that can be applied to the first file to yield the merged output. However, it is usually better to have `diff3` generate the merged output directly; this bypasses some problems with `ed`.

7.1 Selecting Which Changes to Incorporate

You can select all unmerged changes from *older* to *yours* for merging into *mine* with the '`-e`' or '`--ed`' option. You can select only the nonoverlapping unmerged changes with '`-3`' or '`--easy-only`', and you can select only the overlapping changes with '`-x`' or '`--overlap-only`'.

The '`-e`', '`-3`' and '`-x`' options select only *unmerged changes*, i.e. changes where *mine* and *yours* differ; they ignore changes from *older* to *yours* where *mine* and *yours* are identical, because they assume that such changes have already been merged. If this assumption is not a safe

one, you can use the '-A' or '--show-all' option (see Section 7.2 "Marking Conflicts," page 40).

Here is the output of the command `diff3` with each of these three options (see Section 6.1 "Sample diff3 Input," page 35, for the complete contents of the files). Notice that '-e' outputs the union of the disjoint sets of changes output by '-3' and '-x'.

Output of 'diff3 -e lao tzu tao':

```
11a

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
.
8c
  so we may see their result.
.
```

Output of 'diff3 -3 lao tzu tao':

```
8c
  so we may see their result.
.
```

Output of 'diff3 -x lao tzu tao':

```
11a

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
.
```

7.2 Marking Conflicts

`diff3` can mark conflicts in the merged output by bracketing them with special marker lines. A conflict that comes from two files *A* and *B* is marked as follows:

```
<<<<<<< A
lines from A
=======
lines from B
>>>>>>> B
```

A conflict that comes from three files *A*, *B* and *C* is marked as follows:

```
<<<<<<< A
lines from A
||||||| B
lines from B
=======
lines from C
>>>>>>> C
```

The '-A' or '--show-all' option acts like the '-e' option, except that it brackets conflicts, and it outputs all changes from *older* to *yours*, not just the unmerged changes. Thus, given the sample input files (see

Section 6.1 “Sample diff3 Input,” page 35), ‘`diff3 -A lao tzu tao`’ puts brackets around the conflict where only ‘`tzu`’ differs:

```
<<<<<<< tzu
=======
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
>>>>>>> tao
```

And it outputs the three-way conflict as follows:

```
<<<<<<< lao
||||||| tzu
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
=======

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
>>>>>>> tao
```

The ‘`-E`’ or ‘`--show-overlap`’ option outputs less information than the ‘`-A`’ or ‘`--show-all`’ option, because it outputs only unmerged changes, and it never outputs the contents of the second file. Thus the ‘`-E`’ option acts like the ‘`-e`’ option, except that it brackets the first and third files from three-way overlapping changes. Similarly, ‘`-X`’ acts like ‘`-x`’, except it brackets all its (necessarily overlapping) changes. For example, for the three-way overlapping change above, the ‘`-E`’ and ‘`-X`’ options output the following:

```
<<<<<<< lao
=======

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
>>>>>>> tao
```

If you are comparing files that have meaningless or uninformative names, you can use the ‘`-L` *`label`*’ or ‘`--label=`*`label`*’ option to show alternate names in the ‘`<<<<<<<`’, ‘`|||||||`’ and ‘`>>>>>>>`’ brackets. This option can be given up to three times, once for each input file. Thus ‘`diff3 -A -L X -L Y -L Z A B C`’ acts like ‘`diff3 -A A B C`’, except that the output looks like it came from files named ‘`X`’, ‘`Y`’ and ‘`Z`’ rather than from files named ‘`A`’, ‘`B`’ and ‘`C`’.

7.3 Generating the Merged Output Directly

With the ‘`-m`’ or ‘`--merge`’ option, `diff3` outputs the merged file directly. This is more efficient than using `ed` to generate it, and works even with non-text files that `ed` would reject. If you specify ‘`-m`’ without an `ed` script option, ‘`-A`’ (‘`--show-all`’) is assumed.

For example, the command 'diff3 -m lao tzu tao' (see Section 6.1 "Sample diff3 Input," page 35 for a copy of the input files) would output the following:

```
<<<<<<< tzu
=======
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
>>>>>>> tao
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their result.
The two are the same,
But after they are produced,
  they have different names.
<<<<<<< lao
||||||| tzu
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
=======

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
>>>>>>> tao
```

7.4 How diff3 Merges Incomplete Lines

With '-m', incomplete lines (see Chapter 16 "Incomplete Lines," page 81) are simply copied to the output as they are found; if the merged output ends in an conflict and one of the input files ends in an incomplete line, succeeding '|||||||', '=======' or '>>>>>>>' brackets appear somewhere other than the start of a line because they are appended to the incomplete line.

Without '-m', if an ed script option is specified and an incomplete line is found, diff3 generates a warning and acts as if a newline had been present.

7.5 Saving the Changed File

Traditional Unix diff3 generates an ed script without the trailing 'w' and and 'q' commands that save the changes. System V diff3 generates these extra commands. GNU diff3 normally behaves like traditional Unix diff3, but with the '-i' option it behaves like System V diff3 and appends the 'w' and 'q' commands.

The '-i' option requires one of the ed script options '-AeExX3', and is incompatible with the merged output option '-m'.

8 Interactive Merging with `sdiff`

With `sdiff`, you can merge two files interactively based on a side-by-side '`-y`' format comparison (see Section 2.4 "Side by Side," page 16). Use '`-o` *`file`*' or '`--output=`*`file`*' to specify where to put the merged text. See Chapter 15 "Invoking sdiff," page 77, for more details on the options to `sdiff`.

Another way to merge files interactively is to use the Emacs Lisp package `emerge`. See section "emerge" in *The GNU Emacs Manual*, for more information.

8.1 Specifying `diff` Options to `sdiff`

The following `sdiff` options have the same meaning as for `diff`. See Section 12.1 "diff Options," page 57, for the use of these options.

```
-a -b -d -i -t -v
-B -H -I regexp

--ignore-blank-lines  --ignore-case
--ignore-matching-lines=regexp  --ignore-space-change
--left-column  --minimal  --speed-large-files
--suppress-common-lines  --expand-tabs
--text  --version  --width=columns
```

For historical reasons, `sdiff` has alternate names for some options. The '`-l`' option is equivalent to the '`--left-column`' option, and similarly '`-s`' is equivalent to '`--suppress-common-lines`'. The meaning of the `sdiff` '`-w`' and '`-W`' options is interchanged from that of `diff`: with `sdiff`, '`-w` *`columns`*' is equivalent to '`--width=`*`columns`*', and '`-W`' is equivalent to '`--ignore-all-space`'. `sdiff` without the '`-o`' option is equivalent to `diff` with the '`-y`' or '`--side-by-side`' option (see Section 2.4 "Side by Side," page 16).

8.2 Merge Commands

Groups of common lines, with a blank gutter, are copied from the first file to the output. After each group of differing lines, `sdiff` prompts with '`%`' and pauses, waiting for one of the following commands. Follow each command with `RET`.

'`e`' Discard both versions. Invoke a text editor on an empty temporary file, then copy the resulting file to the output.

'`eb`' Concatenate the two versions, edit the result in a temporary file, then copy the edited result to the output.

'el' Edit a copy of the left version, then copy the result to the output.

'er' Edit a copy of the right version, then copy the result to the output.

'l' Copy the left version to the output.

'q' Quit.

'r' Copy the right version to the output.

's' Silently copy common lines.

'v' Verbosely copy common lines. This is the default.

The text editor invoked is specified by the EDITOR environment variable if it is set. The default is system-dependent.

9 Merging with patch

patch takes comparison output produced by diff and applies the differences to a copy of the original file, producing a patched version. With patch, you can distribute just the changes to a set of files instead of distributing the entire file set; your correspondents can apply patch to update their copy of the files with your changes. patch automatically determines the diff format, skips any leading or trailing headers, and uses the headers to determine which file to patch. This lets your correspondents feed an article or message containing a difference listing directly to patch.

patch detects and warns about common problems like forward patches. It saves the original version of the files it patches, and saves any patches that it could not apply. It can also maintain a patchlevel.h file to ensures that your correspondents apply diffs in the proper order.

patch accepts a series of diffs in its standard input, usually separated by headers that specify which file to patch. It applies diff hunks (see Section 1.1 “Hunks,” page 3) one by one. If a hunk does not exactly match the original file, patch uses heuristics to try to patch the file as well as it can. If no approximate match can be found, patch rejects the hunk and skips to the next hunk. patch normally replaces each file *f* with its new version, saving the original file in ‘*f*.orig’, and putting reject hunks (if any) into ‘*f*.rej’.

See Chapter 14 “Invoking patch,” page 69, for detailed information on the options to patch. See Section 14.2 “Backups,” page 70, for more information on how patch names backup files. See Section 14.3 “Rejects,” page 71, for more information on where patch puts reject hunks.

9.1 Selecting the patch Input Format

patch normally determines which diff format the patch file uses by examining its contents. For patch files that contain particularly confusing leading text, you might need to use one of the following options to force patch to interpret the patch file as a certain format of diff. The output formats listed here are the only ones that patch can understand.

‘-c’
‘--context’
: context diff.

‘-e’
‘--ed’
: ed script.

‘-n’
‘--normal’
normal diff.

‘-u’
‘--unified’
unified diff.

9.2 Applying Imperfect Patches

patch tries to skip any leading text in the patch file, apply the diff, and then skip any trailing text. Thus you can feed a news article or mail message directly to patch, and it should work. If the entire diff is indented by a constant amount of white space, patch automatically ignores the indentation.

However, certain other types of imperfect input require user intervention.

9.2.1 Applying Patches with Changed White Space

Sometimes mailers, editors, or other programs change spaces into tabs, or vice versa. If this happens to a patch file or an input file, the files might look the same, but patch will not be able to match them properly. If this problem occurs, use the ‘-l’ or ‘--ignore-white-space’ option, which makes patch compare white space loosely so that any sequence of white space in the patch file matches any sequence of white space in the input files. Non-white-space characters must still match exactly. Each line of the context must still match a line in the input file.

9.2.2 Applying Reversed Patches

Sometimes people run diff with the new file first instead of second. This creates a diff that is “reversed”. To apply such patches, give patch the ‘-R’ or ‘--reverse’ option. patch then attempts to swap each hunk around before applying it. Rejects come out in the swapped format. The ‘-R’ option does not work with ed scripts because there is too little information in them to reconstruct the reverse operation.

Often patch can guess that the patch is reversed. If the first hunk of a patch fails, patch reverses the hunk to see if it can apply it that way. If it can, patch asks you if you want to have the ‘-R’ option set; if it can’t, patch continues to apply the patch normally. This method cannot detect a reversed patch if it is a normal diff and the first command

is an append (which should have been a delete) since appends always succeed, because a null context matches anywhere. But most patches add or change lines rather than delete them, so most reversed normal diffs begin with a delete, which fails, and `patch` notices.

If you apply a patch that you have already applied, `patch` thinks it is a reversed patch and offers to un-apply the patch. This could be construed as a feature. If you did this inadvertently and you don't want to un-apply the patch, just answer '`n`' to this offer and to the subsequent "apply anyway" question—or type `C-c` to kill the `patch` process.

9.2.3 Helping `patch` Find Inexact Matches

For context diffs, and to a lesser extent normal diffs, `patch` can detect when the line numbers mentioned in the patch are incorrect, and it attempts to find the correct place to apply each hunk of the patch. As a first guess, it takes the line number mentioned in the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, `patch` scans both forward and backward for a set of lines matching the context given in the hunk.

First `patch` looks for a place where all lines of the context match. If it cannot find such a place, and it is reading a context or unified diff, and the maximum fuzz factor is set to 1 or more, then `patch` makes another scan, ignoring the first and last line of context. If that fails, and the maximum fuzz factor is set to 2 or more, it makes another scan, ignoring the first two and last two lines of context are ignored. It continues similarly if the maximum fuzz factor is larger.

The '`-F` *`lines`*' or '`--fuzz=`*`lines`*' option sets the maximum fuzz factor to *`lines`*. This option only applies to context and unified diffs; it ignores up to *`lines`* lines while looking for the place to install a hunk. Note that a larger fuzz factor increases the odds of making a faulty patch. The default fuzz factor is 2; it may not be set to more than the number of lines of context in the diff, ordinarily 3.

If `patch` cannot find a place to install a hunk of the patch, it writes the hunk out to a reject file (see Section 14.3 "Rejects," page 71, for information on how reject files are named). It writes out rejected hunks in context format no matter what form the input patch is in. If the input is a normal or `ed` diff, many of the contexts are simply null. The line numbers on the hunks in the reject file may be different from those in the patch file: they show the approximate location where `patch` thinks the failed hunks belong in the new file rather than in the old one.

As it completes each hunk, `patch` tells you whether the hunk succeeded or failed, and if it failed, on which line (in the new file) `patch` thinks the hunk should go. If this is different from the line number

specified in the diff, it tells you the offset. A single large offset *may* indicate that `patch` installed a hunk in the wrong place. `patch` also tells you if it used a fuzz factor to make the match, in which case you should also be slightly suspicious.

`patch` cannot tell if the line numbers are off in an `ed` script, and can only detect wrong line numbers in a normal diff when it finds a change or delete command. It may have the same problem with a context diff using a fuzz factor equal to or greater than the number of lines of context shown in the diff (typically 3). In these cases, you should probably look at a context diff between your original and patched input files to see if the changes make sense. Compiling without errors is a pretty good indication that the patch worked, but not a guarantee.

`patch` usually produces the correct results, even when it must make many guesses. However, the results are guaranteed only when the patch is applied to an exact copy of the file that the patch was generated from.

9.3 Removing Empty Files

Sometimes when comparing two directories, the first directory contains a file that the second directory does not. If you give `diff` the '`-N`' or '`--new-file`' option, it outputs a diff that deletes the contents of this file. By default, `patch` leaves an empty file after applying such a diff. The '`-E`' or '`--remove-empty-files`' option to `patch` deletes output files that are empty after applying the diff.

9.4 Multiple Patches in a File

If the patch file contains more than one patch, `patch` tries to apply each of them as if they came from separate patch files. This means that it determines the name of the file to patch for each patch, and that it examines the leading text before each patch for file names and prerequisite revision level (see Chapter 10 “Making Patches,” page 53, for more on that topic).

For the second and subsequent patches in the patch file, you can give options and another original file name by separating their argument lists with a '`+`'. However, the argument list for a second or subsequent patch may not specify a new patch file, since that does not make sense.

For example, to tell `patch` to strip the first three slashes from the name of the first patch in the patch file and none from subsequent patches, and to use '`code.c`' as the first input file, you can use:

```
patch -p3 code.c + -p0 < patchfile
```

The '-S' or '--skip' option ignores the current patch from the patch file, but continue looking for the next patch in the file. Thus, to ignore the first and third patches in the patch file, you can use:

```
patch -S + + -S + < patch file
```

9.5 Messages and Questions from patch

patch can produce a variety of messages, especially if it has trouble decoding its input. In a few situations where it's not sure how to proceed, patch normally prompts you for more information from the keyboard. There are options to suppress printing non-fatal messages and stopping for keyboard input.

The message 'Hmm...' indicates that patch is reading text in the patch file, attempting to determine whether there is a patch in that text, and if so, what kind of patch it is.

You can inhibit all terminal output from patch, unless an error occurs, by using the '-s', '--quiet', or '--silent' option.

There are two ways you can prevent patch from asking you any questions. The '-f' or '--force' option assumes that you know what you are doing. It assumes the following:

- skip patches that do not contain file names in their headers;
- patch files even though they have the wrong version for the 'Prereq:' line in the patch;
- assume that patches are not reversed even if they look like they are.

The '-t' or '--batch' option is similar to '-f', in that it suppresses questions, but it makes somewhat different assumptions:

- skip patches that do not contain file names in their headers (the same as '-f');
- skip patches for which the file has the wrong version for the 'Prereq:' line in the patch;
- assume that patches are reversed if they look like they are.

patch exits with a non-zero status if it creates any reject files. When applying a set of patches in a loop, you should check the exit status, so you don't apply a later patch to a partially patched file.

10 Tips for Making Patch Distributions

Here are some things you should keep in mind if you are going to distribute patches for updating a software package.

Make sure you have specified the file names correctly, either in a context diff header or with an 'Index:' line. If you are patching files in a subdirectory, be sure to tell the patch user to specify a '-p' or '--strip' option as needed. Take care to not send out reversed patches, since these make people wonder whether they have already applied the patch.

To save people from partially applying a patch before other patches that should have gone before it, you can make the first patch in the patch file update a file with a name like 'patchlevel.h' or 'version.c', which contains a patch level or version number. If the input file contains the wrong version number, patch will complain immediately.

An even clearer way to prevent this problem is to put a 'Prereq:' line before the patch. If the leading text in the patch file contains a line that starts with 'Prereq:', patch takes the next word from that line (normally a version number) and checks whether the next input file contains that word, preceded and followed by either white space or a newline. If not, patch prompts you for confirmation before proceeding. This makes it difficult to accidentally apply patches in the wrong order.

Since patch does not handle incomplete lines properly, make sure that all the source files in your program end with a newline whenever you release a version.

To create a patch that changes an older version of a package into a newer version, first make a copy of the older version in a scratch directory. Typically you do that by unpacking a tar or shar archive of the older version.

You might be able to reduce the size of the patch by renaming or removing some files before making the patch. If the older version of the package contains any files that the newer version does not, or if any files have been renamed between the two versions, make a list of rm and mv commands for the user to execute in the old version directory before applying the patch. Then run those commands yourself in the scratch directory.

If there are any files that you don't need to include in the patch because they can easily be rebuilt from other files (for example, 'TAGS' and output from yacc and makeinfo), replace the versions in the scratch directory with the newer versions, using rm and ln or cp.

Now you can create the patch. The de-facto standard diff format for patch distributions is context format with two lines of context, produced by giving diff the '-C 2' option. Do not use less than two lines

of context, because `patch` typically needs at least two lines for proper operation. Give `diff` the '`-P`' option in case the newer version of the package contains any files that the older one does not. Make sure to specify the scratch directory first and the newer directory second.

Add to the top of the patch a note telling the user any `rm` and `mv` commands to run before applying the patch. Then you can remove the scratch directory.

11 Invoking `cmp`

The `cmp` command compares two files, and if they differ, tells the first byte and line number where they differ. Its arguments are as follows:

```
cmp options... from-file [to-file]
```

The file name '`-`' is always the standard input. `cmp` also uses the standard input if one file name is omitted.

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

11.1 Options to `cmp`

Below is a summary of all of the options that GNU `cmp` accepts. Most options have two equivalent names, one of which is a single letter preceded by '`-`', and the other of which is a long name preceded by '`--`'. Multiple single letter options (unless they take an argument) can be combined into a single command line word: '`-cl`' is equivalent to '`-c -l`'.

'`-c`'
: Print the differing characters. Display control characters as a '`^`' followed by a letter of the alphabet and precede characters that have the high bit set with '`M-`' (which stands for "meta").

'`--ignore-initial=`*bytes*'
: Ignore any differences in the the first *bytes* bytes of the input files. Treat files with fewer than *bytes* bytes as if they are empty.

'`-l`'
: Print the (decimal) offsets and (octal) values of all differing bytes.

'`--print-chars`'
: Print the differing characters. Display control characters as a '`^`' followed by a letter of the alphabet and precede characters that have the high bit set with '`M-`' (which stands for "meta").

'`--quiet`'
'`-s`'
'`--silent`'
: Do not print anything; only return an exit status indicating whether the files differ.

'`--verbose`'
: Print the (decimal) offsets and (octal) values of all differing bytes.

'`-v`'
'`--version`'
: Output the version number of `cmp`.

12 Invoking `diff`

The format for running the `diff` command is:

```
diff options... from-file to-file
```

In the simplest case, `diff` compares the contents of the two files *from-file* and *to-file*. A file name of '-' stands for text read from the standard input. As a special case, 'diff - -' compares a copy of standard input to itself.

If *from-file* is a directory and *to-file* is not, `diff` compares the file in *from-file* whose file name is that of *to-file*, and vice versa. The non-directory file must not be '-'.

If both *from-file* and *to-file* are directories, `diff` compares corresponding files in both directories, in alphabetical order; this comparison is not recursive unless the '-r' or '--recursive' option is given. `diff` never compares the actual contents of a directory as if it were a file. The file that is fully specified may not be standard input, because standard input is nameless and the notion of "file with the same name" does not apply.

`diff` options begin with '-', so normally *from-file* and *to-file* may not begin with '-'. However, '--' as an argument by itself treats the remaining arguments as file names even if they begin with '-'.

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

12.1 Options to `diff`

Below is a summary of all of the options that GNU `diff` accepts. Most options have two equivalent names, one of which is a single letter preceded by '-', and the other of which is a long name preceded by '--'. Multiple single letter options (unless they take an argument) can be combined into a single command line word: '-ac' is equivalent to '-a -c'. Long named options can be abbreviated to any unique prefix of their name. Brackets ([and]) indicate that an option takes an optional argument.

'-*lines*'
: Show *lines* (an integer) lines of context. This option does not specify an output format by itself; it has no effect unless it is combined with '-c' (see Section 2.3.1 "Context Format," page 11) or '-u' (see Section 2.3.2 "Unified Format," page 14). This option is obsolete. For proper operation, `patch` typically needs at least two lines of context.

'-a'
: Treat all files as text and compare them line-by-line, even if they do not seem to be text. See Section 1.7 "Binary," page 6.

'-b' Ignore changes in amount of white space. See Section 1.2 "White Space," page 4.

'-B' Ignore changes that just insert or delete blank lines. See Section 1.3 "Blank Lines," page 5.

'--binary'
Read and write data in binary mode. See Section 1.7 "Binary," page 6.

'--brief' Report only whether the files differ, not the details of the differences. See Section 1.6 "Brief," page 6.

'-c' Use the context output format. See Section 2.3.1 "Context Format," page 11.

'-C *lines*'
'--context[=*lines*]'
Use the context output format, showing *lines* (an integer) lines of context, or three if *lines* is not given. See Section 2.3.1 "Context Format," page 11. For proper operation, `patch` typically needs at least two lines of context.

'--changed-group-format=*format*'
Use *format* to output a line group containing differing lines from both files in if-then-else format. See Section 2.7.1 "Line Group Formats," page 21.

'-d' Change the algorithm perhaps find a smaller set of changes. This makes `diff` slower (sometimes much slower). See Chapter 5 "diff Performance," page 33.

'-D *name*' Make merged '#ifdef' format output, conditional on the preprocessor macro *name*. See Section 2.7 "If-then-else," page 21.

'-e'
'--ed' Make output that is a valid `ed` script. See Section 2.6.1 "ed Scripts," page 18.

'--exclude=*pattern*'
When comparing directories, ignore files and subdirectories whose basenames match *pattern*. See Chapter 3 "Comparing Directories," page 29.

'--exclude-from=*file*'
When comparing directories, ignore files and subdirectories whose basenames match any pattern contained in *file*. See Chapter 3 "Comparing Directories," page 29.

'`--expand-tabs`'
: Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See Section 4.1 "Tabs," page 31.

'`-f`'
: Make output that looks vaguely like an `ed` script but has changes in the order they appear in the file. See Section 2.6.2 "Forward ed," page 20.

'`-F` *regexp*'
: In context and unified format, for each hunk of differences, show some of the last preceding line that matches *regexp*. See Section 2.3.3.1 "Specified Headings," page 15.

'`--forward-ed`'
: Make output that looks vaguely like an `ed` script but has changes in the order they appear in the file. See Section 2.6.2 "Forward ed," page 20.

'`-h`'
: This option currently has no effect; it is present for Unix compatibility.

'`-H`'
: Use heuristics to speed handling of large files that have numerous scattered small changes. See Chapter 5 "diff Performance," page 33.

'`--horizon-lines=`*lines*'
: Do not discard the last *lines* lines of the common prefix and the first *lines* lines of the common suffix. See Chapter 5 "diff Performance," page 33.

'`-i`'
: Ignore changes in case; consider upper- and lower-case letters equivalent. See Section 1.4 "Case Folding," page 5.

'`-I` *regexp*'
: Ignore changes that just insert or delete lines that match *regexp*. See Section 1.5 "Specified Folding," page 5.

'`--ifdef=`*name*'
: Make merged if-then-else output using *name*. See Section 2.7 "If-then-else," page 21.

'`--ignore-all-space`'
: Ignore white space when comparing lines. See Section 1.2 "White Space," page 4.

'`--ignore-blank-lines`'
: Ignore changes that just insert or delete blank lines. See Section 1.3 "Blank Lines," page 5.

'--ignore-case'
: Ignore changes in case; consider upper- and lower-case to be the same. See Section 1.4 "Case Folding," page 5.

'--ignore-matching-lines=*regexp*'
: Ignore changes that just insert or delete lines that match *regexp*. See Section 1.5 "Specified Folding," page 5.

'--ignore-space-change'
: Ignore changes in amount of white space. See Section 1.2 "White Space," page 4.

'--initial-tab'
: Output a tab rather than a space before the text of a line in normal or context format. This causes the alignment of tabs in the line to look normal. See Section 4.1 "Tabs," page 31.

'-l'
: Pass the output through `pr` to paginate it. See Section 4.2 "Pagination," page 31.

'-L *label*'
: Use *label* instead of the file name in the context format (see Section 2.3.1 "Context Format," page 11) and unified format (see Section 2.3.2 "Unified Format," page 14) headers. See Section 2.6.3 "RCS," page 20.

'--label=*label*'
: Use *label* instead of the file name in the context format (see Section 2.3.1 "Context Format," page 11) and unified format (see Section 2.3.2 "Unified Format," page 14) headers.

'--left-column'
: Print only the left column of two common lines in side by side format. See Section 2.5 "Side by Side Format," page 17.

'--line-format=*format*'
: Use *format* to output all input lines in if-then-else format. See Section 2.7.2 "Line Formats," page 24.

'--minimal'
: Change the algorithm to perhaps find a smaller set of changes. This makes `diff` slower (sometimes much slower). See Chapter 5 "diff Performance," page 33.

'-n'
: Output RCS-format diffs; like '-f' except that each command specifies the number of lines affected. See Section 2.6.3 "RCS," page 20.

'-N'
'--new-file'
: In directory comparison, if a file is found in only one directory, treat it as present but empty in the other directory. See Chapter 3 "Comparing Directories," page 29.

'`--new-group-format=`*`format`*'
: Use *`format`* to output a group of lines taken from just the second file in if-then-else format. See Section 2.7.1 "Line Group Formats," page 21.

'`--new-line-format=`*`format`*'
: Use *`format`* to output a line taken from just the second file in if-then-else format. See Section 2.7.2 "Line Formats," page 24.

'`--old-group-format=`*`format`*'
: Use *`format`* to output a group of lines taken from just the first file in if-then-else format. See Section 2.7.1 "Line Group Formats," page 21.

'`--old-line-format=`*`format`*'
: Use *`format`* to output a line taken from just the first file in if-then-else format. See Section 2.7.2 "Line Formats," page 24.

'`-p`' Show which C function each change is in. See Section 2.3.3.2 "C Function Headings," page 16.

'`-P`' When comparing directories, if a file appears only in the second directory of the two, treat it as present but empty in the other. See Chapter 3 "Comparing Directories," page 29.

'`--paginate`'
: Pass the output through `pr` to paginate it. See Section 4.2 "Pagination," page 31.

'`-q`' Report only whether the files differ, not the details of the differences. See Section 1.6 "Brief," page 6.

'`-r`' When comparing directories, recursively compare any subdirectories found. See Chapter 3 "Comparing Directories," page 29.

'`--rcs`' Output RCS-format diffs; like '`-f`' except that each command specifies the number of lines affected. See Section 2.6.3 "RCS," page 20.

'`--recursive`'
: When comparing directories, recursively compare any subdirectories found. See Chapter 3 "Comparing Directories," page 29.

'`--report-identical-files`'
: Report when two files are the same. See Chapter 3 "Comparing Directories," page 29.

'`-s`' Report when two files are the same. See Chapter 3 "Comparing Directories," page 29.

'-S *file*' When comparing directories, start with the file *file*. This is used for resuming an aborted comparison. See Chapter 3 "Comparing Directories," page 29.

'--sdiff-merge-assist'
Print extra information to help sdiff. sdiff uses this option when it runs diff. This option is not intended for users to use directly.

'--show-c-function'
Show which C function each change is in. See Section 2.3.3.2 "C Function Headings," page 16.

'--show-function-line=*regexp*'
In context and unified format, for each hunk of differences, show some of the last preceding line that matches *regexp*. See Section 2.3.3.1 "Specified Headings," page 15.

'--side-by-side'
Use the side by side output format. See Section 2.5 "Side by Side Format," page 17.

'--speed-large-files'
Use heuristics to speed handling of large files that have numerous scattered small changes. See Chapter 5 "diff Performance," page 33.

'--starting-file=*file*'
When comparing directories, start with the file *file*. This is used for resuming an aborted comparison. See Chapter 3 "Comparing Directories," page 29.

'--suppress-common-lines'
Do not print common lines in side by side format. See Section 2.5 "Side by Side Format," page 17.

'-t' Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See Section 4.1 "Tabs," page 31.

'-T' Output a tab rather than a space before the text of a line in normal or context format. This causes the alignment of tabs in the line to look normal. See Section 4.1 "Tabs," page 31.

'--text' Treat all files as text and compare them line-by-line, even if they do not appear to be text. See Section 1.7 "Binary," page 6.

'-u' Use the unified output format. See Section 2.3.2 "Unified Format," page 14.

‘`--unchanged-group-format=`*`format`*’
: Use *`format`* to output a group of common lines taken from both files in if-then-else format. See Section 2.7.1 “Line Group Formats,” page 21.

‘`--unchanged-line-format=`*`format`*’
: Use *`format`* to output a line common to both files in if-then-else format. See Section 2.7.2 “Line Formats,” page 24.

‘`--unidirectional-new-file`’
: When comparing directories, if a file appears only in the second directory of the two, treat it as present but empty in the other. See Chapter 3 “Comparing Directories,” page 29.

‘`-U` *`lines`*’
‘`--unified`[`=`*`lines`*]’
: Use the unified output format, showing *`lines`* (an integer) lines of context, or three if *`lines`* is not given. See Section 2.3.2 “Unified Format,” page 14. For proper operation, `patch` typically needs at least two lines of context.

‘`-v`’
‘`--version`’
: Output the version number of `diff`.

‘`-w`’
: Ignore white space when comparing lines. See Section 1.2 “White Space,” page 4.

‘`-W` *`columns`*’
‘`--width=`*`columns`*’
: Use an output width of *`columns`* in side by side format. See Section 2.5 “Side by Side Format,” page 17.

‘`-x` *`pattern`*’
: When comparing directories, ignore files and subdirectories whose basenames match *`pattern`*. See Chapter 3 “Comparing Directories,” page 29.

‘`-X` *`file`*’
: When comparing directories, ignore files and subdirectories whose basenames match any pattern contained in *`file`*. See Chapter 3 “Comparing Directories,” page 29.

‘`-y`’
: Use the side by side output format. See Section 2.5 “Side by Side Format,” page 17.

13 Invoking `diff3`

The `diff3` command compares three files and outputs descriptions of their differences. Its arguments are as follows:

```
diff3 options... mine older yours
```

The files to compare are *mine*, *older*, and *yours*. At most one of these three file names may be '-', which tells `diff3` to read the standard input for that file.

An exit status of 0 means `diff3` was successful, 1 means some conflicts were found, and 2 means trouble.

13.1 Options to `diff3`

Below is a summary of all of the options that GNU `diff3` accepts. Multiple single letter options (unless they take an argument) can be combined into a single command line argument.

'`-a`'
: Treat all files as text and compare them line-by-line, even if they do not appear to be text. See Section 1.7 "Binary," page 6.

'`-A`'
: Incorporate all changes from *older* to *yours* into *mine*, surrounding all conflicts with bracket lines. See Section 7.2 "Marking Conflicts," page 40.

'`-e`'
: Generate an `ed` script that incorporates all the changes from *older* to *yours* into *mine*. See Section 7.1 "Which Changes," page 39.

'`-E`'
: Like '`-e`', except bracket lines from overlapping changes' first and third files. See Section 7.2 "Marking Conflicts," page 40. With '`-e`', an overlapping change looks like this:

```
<<<<<<< mine
lines from mine
=======
lines from yours
>>>>>>> yours
```

'`--ed`'
: Generate an `ed` script that incorporates all the changes from *older* to *yours* into *mine*. See Section 7.1 "Which Changes," page 39.

'`--easy-only`'
: Like '`-e`', except output only the nonoverlapping changes. See Section 7.1 "Which Changes," page 39.

'`-i`'
: Generate '`w`' and '`q`' commands at the end of the `ed` script for System V compatibility. This option must be combined with

one of the '-AeExX3' options, and may not be combined with '-m'. See Section 7.5 "Saving the Changed File," page 42.

'--initial-tab'
: Output a tab rather than two spaces before the text of a line in normal format. This causes the alignment of tabs in the line to look normal. See Section 4.1 "Tabs," page 31.

'-L *label*'
'--label=*label*'
: Use the label *label* for the brackets output by the '-A', '-E' and '-X' options. This option may be given up to three times, one for each input file. The default labels are the names of the input files. Thus 'diff3 -L X -L Y -L Z -m A B C' acts like 'diff3 -m A B C', except that the output looks like it came from files named 'X', 'Y' and 'Z' rather than from files named 'A', 'B' and 'C'. See Section 7.2 "Marking Conflicts," page 40.

'-m'
'--merge'
: Apply the edit script to the first file and send the result to standard output. Unlike piping the output from diff3 to ed, this works even for binary files and incomplete lines. '-A' is assumed if no edit script option is specified. See Section 7.3 "Bypassing ed," page 41.

'--overlap-only'
: Like '-e', except output only the overlapping changes. See Section 7.1 "Which Changes," page 39.

'--show-all'
: Incorporate all unmerged changes from *older* to *yours* into *mine*, surrounding all overlapping changes with bracket lines. See Section 7.2 "Marking Conflicts," page 40.

'--show-overlap'
: Like '-e', except bracket lines from overlapping changes' first and third files. See Section 7.2 "Marking Conflicts," page 40.

'-T'
: Output a tab rather than two spaces before the text of a line in normal format. This causes the alignment of tabs in the line to look normal. See Section 4.1 "Tabs," page 31.

'--text'
: Treat all files as text and compare them line-by-line, even if they do not appear to be text. See Section 1.7 "Binary," page 6.

'-v'
'--version'
: Output the version number of diff3.

'-x' Like '-e', except output only the overlapping changes. See Section 7.1 "Which Changes," page 39.

'-X' Like '-E', except output only the overlapping changes. In other words, like '-x', except bracket changes as in '-E'. See Section 7.2 "Marking Conflicts," page 40.

'-3' Like '-e', except output only the nonoverlapping changes. See Section 7.1 "Which Changes," page 39.

14 Invoking patch

Normally patch is invoked like this:

```
patch <patchfile
```

The full format for invoking patch is:

```
patch options... [origfile [patchfile]] [+ options... [origfile]]...
```

If you do not specify *patchfile*, or if *patchfile* is '-', patch reads the patch (that is, the diff output) from the standard input.

You can specify one or more of the original files as *orig* arguments; each one and options for interpreting it is separated from the others with a '+'. See Section 9.4 "Multiple Patches," page 50, for more information.

If you do not specify an input file on the command line, patch tries to figure out from the *leading text* (any text in the patch that comes before the diff output) which file to edit. In the header of a context or unified diff, patch looks in lines beginning with '***', '---', or '+++'; among those, it chooses the shortest name of an existing file. Otherwise, if there is an 'Index:' line in the leading text, patch tries to use the file name from that line. If patch cannot figure out the name of an existing file from the leading text, it prompts you for the name of the file to patch.

If the input file does not exist or is read-only, and a suitable RCS or SCCS file exists, patch attempts to check out or get the file before proceeding.

By default, patch replaces the original input file with the patched version, after renaming the original file into a backup file (see Section 14.2 "Backups," page 70, for a description of how patch names backup files). You can also specify where to put the output with the '-o *output-file*' or '--output=*output-file*' option.

14.1 Applying Patches in Other Directories

The '-d *directory*' or '--directory=*directory*' option to patch makes directory *directory* the current directory for interpreting both file names in the patch file, and file names given as arguments to other options (such as '-B' and '-o'). For example, while in a news reading program, you can patch a file in the '/usr/src/emacs' directory directly from the article containing the patch like this:

```
| patch -d /usr/src/emacs
```

Sometimes the file names given in a patch contain leading directories, but you keep your files in a directory different from the one given in the patch. In those cases, you can use the '-p[*number*]' or '--strip[=*number*]' option to set the file name strip count to *number*. The strip count tells patch how many slashes, along with the directory names between them,

to strip from the front of file names. '-p' with no *number* given is equivalent to '-p0'. By default, patch strips off all leading directories, leaving just the base file names, except that when a file name given in the patch is a relative file name and all of its leading directories already exist, patch does not strip off the leading directory. (A *relative* file name is one that does not start with a slash.)

patch looks for each file (after any slashes have been stripped) in the current directory, or if you used the '-d *directory*' option, in that directory.

For example, suppose the file name in the patch file is '/gnu/src/emacs/etc/NEWS'. Using '-p' or '-p0' gives the entire file name unmodified, '-p1' gives 'gnu/src/emacs/etc/NEWS' (no leading slash), '-p4' gives 'etc/NEWS', and not specifying '-p' at all gives 'NEWS'.

14.2 Backup File Names

Normally, patch renames an original input file into a backup file by appending to its name the extension '.orig', or '~' on systems that do not support long file names. The '-b *backup-suffix*' or '--suffix=*backup-suffix*' option uses *backup-suffix* as the backup extension instead.

Alternately, you can specify the extension for backup files with the SIMPLE_BACKUP_SUFFIX environment variable, which the options override.

patch can also create numbered backup files the way GNU Emacs does. With this method, instead of having a single backup of each file, patch makes a new backup file name each time it patches a file. For example, the backups of a file named 'sink' would be called, successively, 'sink.~1~', 'sink.~2~', 'sink.~3~', etc.

The '-V *backup-style*' or '--version-control=*backup-style*' option takes as an argument a method for creating backup file names. You can alternately control the type of backups that patch makes with the VERSION_CONTROL environment variable, which the '-V' option overrides. The value of the VERSION_CONTROL environment variable and the argument to the '-V' option are like the GNU Emacs version-control variable (see Section 14.2 "The GNU Emacs Manual," page 70, for more information on backup versions in Emacs). They also recognize synonyms that are more descriptive. The valid values are listed below; unique abbreviations are acceptable.

't'
'numbered'
    Always make numbered backups.

'nil'
'existing'
: Make numbered backups of files that already have them, simple backups of the others. This is the default.

'never'
'simple'
: Always make simple backups.

Alternately, you can tell patch to prepend a prefix, such as a directory name, to produce backup file names. The '-B *backup-prefix*' or '--prefix=*backup-prefix*' option makes backup files by prepending *backup-prefix* to them. If you use this option, patch ignores any '-b' option that you give.

If the backup file already exists, patch creates a new backup file name by changing the first lowercase letter in the last component of the file name into uppercase. If there are no more lowercase letters in the name, it removes the first character from the name. It repeats this process until it comes up with a backup file name that does not already exist.

If you specify the output file with the '-o' option, that file is the one that is backed up, not the input file.

14.3 Reject File Names

The names for reject files (files containing patches that patch could not find a place to apply) are normally the name of the output file with '.rej' appended (or '#' on systems that do not support long file names).

Alternatively, you can tell patch to place all of the rejected patches in a single file. The '-r *reject-file*' or '--reject-file=*reject-file*' option uses *reject-file* as the reject file name.

14.4 Options to patch

Here is a summary of all of the options that patch accepts. Older versions of patch do not accept long-named options or the '-t', '-E', or '-V' options.

Multiple single-letter options that do not take an argument can be combined into a single command line argument (with only one dash). Brackets ([and]) indicate that an option takes an optional argument.

'-b *backup-suffix*'
: Use *backup-suffix* as the backup extension instead of '.orig' or '~'. See Section 14.2 "Backups," page 70.

'-B *backup-prefix*'
: Use *backup-prefix* as a prefix to the backup file name. If this option is specified, any '-b' option is ignored. See Section 14.2 "Backups," page 70.

'--batch'
: Do not ask any questions. See Section 9.5 "patch Messages," page 51.

'-c'
'--context'
: Interpret the patch file as a context diff. See Section 9.1 "patch Input," page 47.

'-d *directory*'
'--directory=*directory*'
: Makes directory *directory* the current directory for interpreting both file names in the patch file, and file names given as arguments to other options. See Section 14.1 "patch Directories," page 69.

'-D *name*'
: Make merged if-then-else output using *format*. See Section 2.7 "If-then-else," page 21.

'--debug=*number*'
: Set internal debugging flags. Of interest only to patch patchers.

'-e'
'--ed'
: Interpret the patch file as an ed script. See Section 9.1 "patch Input," page 47.

'-E'
: Remove output files that are empty after the patches have been applied. See Section 9.3 "Empty Files," page 50.

'-f'
: Assume that the user knows exactly what he or she is doing, and do not ask any questions. See Section 9.5 "patch Messages," page 51.

'-F *lines*'
: Set the maximum fuzz factor to *lines*. See Section 9.2.3 "Inexact," page 49.

'--force'
: Assume that the user knows exactly what he or she is doing, and do not ask any questions. See Section 9.5 "patch Messages," page 51.

'--forward'
: Ignore patches that patch thinks are reversed or already applied. See also '-R'. See Section 9.2.2 "Reversed Patches," page 48.

'`--fuzz=`*`lines`*'
: Set the maximum fuzz factor to *`lines`*. See Section 9.2.3 "Inexact," page 49.

'`--help`'
: Print a summary of the options that `patch` recognizes, then exit.

'`--ifdef=`*`name`*'
: Make merged if-then-else output using *`format`*. See Section 2.7 "If-then-else," page 21.

'`--ignore-white-space`'
'`-l`'
: Let any sequence of white space in the patch file match any sequence of white space in the input file. See Section 9.2.1 "Changed White Space," page 48.

'`-n`'
'`--normal`'
: Interpret the patch file as a normal diff. See Section 9.1 "patch Input," page 47.

'`-N`'
: Ignore patches that `patch` thinks are reversed or already applied. See also '`-R`'. See Section 9.2.2 "Reversed Patches," page 48.

'`-o` *`output-file`*'
'`--output=`*`output-file`*'
: Use *`output-file`* as the output file name. See Section 14.4 "patch Options," page 71.

'`-p`[*`number`*]'
: Set the file name strip count to *`number`*. See Section 14.1 "patch Directories," page 69.

'`--prefix=`*`backup-prefix`*'
: Use *`backup-prefix`* as a prefix to the backup file name. If this option is specified, any '`-b`' option is ignored. See Section 14.2 "Backups," page 70.

'`--quiet`'
: Work silently unless an error occurs. See Section 9.5 "patch Messages," page 51.

'`-r` *`reject-file`*'
: Use *`reject-file`* as the reject file name. See Section 14.3 "Rejects," page 71.

'`-R`'
: Assume that this patch was created with the old and new files swapped. See Section 9.2.2 "Reversed Patches," page 48.

'`--reject-file=`*`reject-file`*'
: Use *`reject-file`* as the reject file name. See Section 14.3 "Rejects," page 71.

'--remove-empty-files'
: Remove output files that are empty after the patches have been applied. See Section 9.3 "Empty Files," page 50.

'--reverse'
: Assume that this patch was created with the old and new files swapped. See Section 9.2.2 "Reversed Patches," page 48.

'-s'
: Work silently unless an error occurs. See Section 9.5 "patch Messages," page 51.

'-S'
: Ignore this patch from the patch file, but continue looking for the next patch in the file. See Section 9.4 "Multiple Patches," page 50.

'--silent'
: Work silently unless an error occurs. See Section 9.5 "patch Messages," page 51.

'--skip'
: Ignore this patch from the patch file, but continue looking for the next patch in the file. See Section 9.4 "Multiple Patches," page 50.

'--strip[=*number*]'
: Set the file name strip count to *number*. See Section 14.1 "patch Directories," page 69.

'--suffix=*backup-suffix*'
: Use *backup-suffix* as the backup extension instead of '.orig' or '~'. See Section 14.2 "Backups," page 70.

'-t'
: Do not ask any questions. See Section 9.5 "patch Messages," page 51.

'-u'
'--unified'
: Interpret the patch file as a unified diff. See Section 9.1 "patch Input," page 47.

'-v'
: Output the revision header and patch level of patch.

'-V *backup-style*'
: Select the kind of backups to make. See Section 14.2 "Backups," page 70.

'--version'
: Output the revision header and patch level of patch, then exit.

'--version=control=*backup-style*'
: Select the kind of backups to make. See Section 14.2 "Backups," page 70.

'-x *number*'
: Set internal debugging flags. Of interest only to patch patchers.

15 Invoking `sdiff`

The `sdiff` command merges two files and interactively outputs the results. Its arguments are as follows:

```
sdiff -o outfile options... from-file to-file
```

This merges *from-file* with *to-file*, with output to *outfile*. If *from-file* is a directory and *to-file* is not, `sdiff` compares the file in *from-file* whose file name is that of *to-file*, and vice versa. *from-file* and *to-file* may not both be directories.

`sdiff` options begin with '`-`', so normally *from-file* and *to-file* may not begin with '`-`'. However, '`--`' as an argument by itself treats the remaining arguments as file names even if they begin with '`-`'. You may not use '`-`' as an input file.

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

`sdiff` without '`-o`' (or '`--output`') produces a side-by-side difference. This usage is obsolete; use '`diff --side-by-side`' instead.

15.1 Options to `sdiff`

Below is a summary of all of the options that GNU `sdiff` accepts. Each option has two equivalent names, one of which is a single letter preceded by '`-`', and the other of which is a long name preceded by '`--`'. Multiple single letter options (unless they take an argument) can be combined into a single command line argument. Long named options can be abbreviated to any unique prefix of their name.

'`-a`' Treat all files as text and compare them line-by-line, even if they do not appear to be text. See Section 1.7 "Binary," page 6.

'`-b`' Ignore changes in amount of white space. See Section 1.2 "White Space," page 4.

'`-B`' Ignore changes that just insert or delete blank lines. See Section 1.3 "Blank Lines," page 5.

'`-d`' Change the algorithm to perhaps find a smaller set of changes. This makes `sdiff` slower (sometimes much slower). See Chapter 5 "diff Performance," page 33.

'`-H`' Use heuristics to speed handling of large files that have numerous scattered small changes. See Chapter 5 "diff Performance," page 33.

'--expand-tabs'
: Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See Section 4.1 "Tabs," page 31.

'-i'
: Ignore changes in case; consider upper- and lower-case to be the same. See Section 1.4 "Case Folding," page 5.

'-I *regexp*'
: Ignore changes that just insert or delete lines that match *regexp*. See Section 1.5 "Specified Folding," page 5.

'--ignore-all-space'
: Ignore white space when comparing lines. See Section 1.2 "White Space," page 4.

'--ignore-blank-lines'
: Ignore changes that just insert or delete blank lines. See Section 1.3 "Blank Lines," page 5.

'--ignore-case'
: Ignore changes in case; consider upper- and lower-case to be the same. See Section 1.4 "Case Folding," page 5.

'--ignore-matching-lines=*regexp*'
: Ignore changes that just insert or delete lines that match *regexp*. See Section 1.5 "Specified Folding," page 5.

'--ignore-space-change'
: Ignore changes in amount of white space. See Section 1.2 "White Space," page 4.

'-l'
'--left-column'
: Print only the left column of two common lines. See Section 2.5 "Side by Side Format," page 17.

'--minimal'
: Change the algorithm to perhaps find a smaller set of changes. This makes `sdiff` slower (sometimes much slower). See Chapter 5 "diff Performance," page 33.

'-o *file*'
'--output=*file*'
: Put merged output into *file*. This option is required for merging.

'-s'
'--suppress-common-lines'
: Do not print common lines. See Section 2.5 "Side by Side Format," page 17.

'--speed-large-files'
: Use heuristics to speed handling of large files that have numerous scattered small changes. See Chapter 5 "diff Performance," page 33.

'-t'
: Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See Section 4.1 "Tabs," page 31.

'--text'
: Treat all files as text and compare them line-by-line, even if they do not appear to be text. See Section 1.7 "Binary," page 6.

'-v'
'--version'
: Output the version number of `sdiff`.

'-w *columns*'
'--width=*columns*'
: Use an output width of *columns*. See Section 2.5 "Side by Side Format," page 17. Note that for historical reasons, this option is '-W' in `diff`, '-w' in `sdiff`.

'-W'
: Ignore horizontal white space when comparing lines. See Section 1.2 "White Space," page 4. Note that for historical reasons, this option is '-w' in `diff`, '-W' in `sdiff`.

16 Incomplete Lines

When an input file ends in a non-newline character, its last line is called an *incomplete line* because its last character is not a newline. All other lines are called *full lines* and end in a newline character. Incomplete lines do not match full lines unless differences in white space are ignored (see Section 1.2 "White Space," page 4).

An incomplete line is normally distinguished on output from a full line by a following line that starts with '\'. However, the RCS format (see Section 2.6.3 "RCS," page 20) outputs the incomplete line as-is, without any trailing newline or following line. The side by side format normally represents incomplete lines as-is, but in some cases uses a '\' or '/' gutter marker; See Section 2.4 "Side by Side," page 16. The if-then-else line format preserves a line's incompleteness with '%L', and discards the newline with '%l'; See Section 2.7.2 "Line Formats," page 24. Finally, with the `ed` and forward `ed` output formats (see Chapter 2 "Output Formats," page 9) `diff` cannot represent an incomplete line, so it pretends there was a newline and reports an error.

For example, suppose 'F' and 'G' are one-byte files that contain just 'f' and 'g', respectively. Then 'diff F G' outputs

```
1c1
< f
\ No newline at end of file
---
> g
\ No newline at end of file
```

(The exact message may differ in non-English locales.) 'diff -n F G' outputs the following without a trailing newline:

```
d1 1
a1 1
g
```

'diff -e F G' reports two errors and outputs the following:

```
1c
g
.
```

17 Future Projects

Here are some ideas for improving GNU `diff` and `patch`. The GNU project has identified some improvements as potential programming projects for volunteers. You can also help by reporting any bugs that you find.

If you are a programmer and would like to contribute something to the GNU project, please consider volunteering for one of these projects. If you are seriously contemplating work, please write to '`gnu@prep.ai.mit.edu`' to coordinate with other volunteers.

17.1 Suggested Projects for Improving GNU `diff` and `patch`

One should be able to use GNU `diff` to generate a patch from any pair of directory trees, and given the patch and a copy of one such tree, use `patch` to generate a faithful copy of the other. Unfortunately, some changes to directory trees cannot be expressed using current patch formats; also, `patch` does not handle some of the existing formats. These shortcomings motivate the following suggested projects.

17.1.1 Handling Changes to the Directory Structure

`diff` and `patch` do not handle some changes to directory structure. For example, suppose one directory tree contains a directory named '`D`' with some subsidiary files, and another contains a file with the same name '`D`'. '`diff -r`' does not output enough information for `patch` to transform the the directory subtree into the file.

There should be a way to specify that a file has been deleted without having to include its entire contents in the patch file. There should also be a way to tell `patch` that a file was renamed, even if there is no way for `diff` to generate such information.

These problems can be fixed by extending the `diff` output format to represent changes in directory structure, and extending `patch` to understand these extensions.

17.1.2 Files that are Neither Directories Nor Regular Files

Some files are neither directories nor regular files: they are unusual files like symbolic links, device special files, named pipes, and sockets. Currently, `diff` treats symbolic links like regular files; it treats other special files like regular files if they are specified at the top level, but

simply reports their presence when comparing directories. This means that `patch` cannot represent changes to such files. For example, if you change which file a symbolic link points to, `diff` outputs the difference between the two files, instead of the change to the symbolic link.

`diff` should optionally report changes to special files specially, and `patch` should be extended to understand these extensions.

17.1.3 File Names that Contain Unusual Characters

When a file name contains an unusual character like a newline or white space, '`diff -r`' generates a patch that `patch` cannot parse. The problem is with format of `diff` output, not just with `patch`, because with odd enough file names one can cause `diff` to generate a patch that is syntactically correct but patches the wrong files. The format of `diff` output should be extended to handle all possible file names.

17.1.4 Arbitrary Limits

GNU `diff` can analyze files with arbitrarily long lines and files that end in incomplete lines. However, `patch` cannot patch such files. The `patch` internal limits on line lengths should be removed, and `patch` should be extended to parse `diff` reports of incomplete lines.

17.1.5 Handling Files that Do Not Fit in Memory

`diff` operates by reading both files into memory. This method fails if the files are too large, and `diff` should have a fallback.

One way to do this is to scan the files sequentially to compute hash codes of the lines and put the lines in equivalence classes based only on hash code. Then compare the files normally. This does produce some false matches.

Then scan the two files sequentially again, checking each match to see whether it is real. When a match is not real, mark both the "matching" lines as changed. Then build an edit script as usual.

The output routines would have to be changed to scan the files sequentially looking for the text to print.

17.1.6 Ignoring Certain Changes

It would be nice to have a feature for specifying two strings, one in *from-file* and one in *to-file*, which should be considered to match. Thus, if the two strings are '`foo`' and '`bar`', then if two lines differ only

in that 'foo' in file 1 corresponds to 'bar' in file 2, the lines are treated as identical.

It is not clear how general this feature can or should be, or what syntax should be used for it.

17.2 Reporting Bugs

If you think you have found a bug in GNU cmp, diff, diff3, sdiff, or patch, please report it by electronic mail to 'bug-gnu-utils@prep.ai.mit.edu'. Send as precise a description of the problem as you can, including sample input files that produce the bug, if applicable.

Because Larry Wall has not released a new version of patch since mid 1988 and the GNU version of patch has been changed since then, please send bug reports for patch by electronic mail to both 'bug-gnu-utils@prep.ai.mit.edu' and 'lwall@netlabs.com'.

Concept Index

!

'`!`' output format 11

+

'`+-`' output format 14

<

'`<`' output format 9
'`<<<<<<<`' for marking conflicts 40

A

aligning tabstops 31
alternate file names 16

B

backup file names 70
binary file diff 6
binary file patching 84
blank and tab difference suppression 4
blank line difference suppression 5
brief difference reports 6
bug reports 85

C

C function headings 16
C if-then-else output format 21
case difference suppression 5
`cmp` invocation 55
`cmp` options 55
columnar output 16
comparing three files 35
conflict 39
conflict marking 40
context output format 11

D

diagnostics from `patch` 51
`diff` invocation 57
diff merging 45
`diff` options 57
`diff` sample input 9
`diff3` hunks 36
`diff3` invocation 65
`diff3` options 65
`diff3` sample input 35
directories and patch 69
directory structure changes 83

E

`ed` script output format 18
empty files, removing 50

F

file name alternates 16
file names with unusual characters 84
format of `diff` output 9
format of `diff3` output 35
formats for if-then-else line groups 21
forward `ed` script output format 20
full lines 81
function headings, C 16
fuzz factor when patching 49

H

headings 15
hunks 3
hunks for `diff3` 36

I

if-then-else output format 21
`ifdef` output format 21
imperfect patch application 48
incomplete line merging 42
incomplete lines 81
inexact patches 49
interactive merging 45
introduction 3
invoking `cmp` 55
invoking `diff` 57
invoking `diff3` 65
invoking `patch` 69
invoking `sdiff` 77

L

large files 84
line formats 24
line group formats 21

M

merge commands 45
merged diff3 format 41
merged output format 21
merging from a common ancestor 39
merging interactively 45
messages from patch 51
multiple patches 50

N

newline treatment by diff 81
normal output format 9

O

options for cmp 55
options for diff 57
options for diff3 65
options for patch 71
options for sdiff 77
output formats 9
overlap 39
overlapping change, selection of 39
overview of diff and patch 1

P

paginating diff output 31
patch input format 47
patch invocation 69
patch making tips 53
patch messages and questions 51
patch options 71
patching directories 69
performance of diff 33
projects for directories 83

R

RCS script output format 20
regular expression matching headings 15
regular expression suppression 5
reject file names 71
removing empty files 50
reporting bugs 85
reversed patches 48

S

sample input for diff 9
sample input for diff3 35
script output formats 18
sdiff invocation 77
sdiff options 77
sdiff output format 45
section headings 15
side by side 16
side by side format 17
special files 83
specified headings 15
summarizing which files differ 6
System V diff3 compatibility 42

T

tab and blank difference suppression . . . 4
tabstop alignment 31
text versus binary diff 6
tips for patch making 53
two-column output 16

U

unified output format 14
unmerged change 39

W

white space in patches 48

Using GNU CC

Richard M. Stallman

Last updated 19 September 1994

for version 2.6

Copyright © 1988, 89, 92, 93, 94, 1995 Free Software Foundation, Inc.

For GCC Version 2.6.
ISBN 1-882114-35-3

Published by the Free Software Foundation
675 Massachusetts Avenue
Cambridge, MA 02139 USA

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "GNU General Public License," "Funding for Free Software," and "Protect Your Freedom—Fight 'Look And Feel'" are included exactly as in the original, and provided 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 manual into another language, under the above conditions for modified versions, except that the sections entitled "GNU General Public License," "Funding for Free Software," and "Protect Your Freedom—Fight 'Look And Feel'", and this permission notice, may be included in translations approved by the Free Software Foundation instead of in the original English.

Short Contents

GNU GENERAL PUBLIC LICENSE 1
Contributors to GNU CC 11
1 Funding Free Software 15
2 Protect Your Freedom—Fight “Look And Feel” 17
3 Compile C, C++, or Objective C 21
4 GNU CC Command Options 23
5 Installing GNU CC 103
6 Extensions to the C Language Family 141
7 Extensions to the C++ Language 189
8 `gcov`: a Test Coverage Program 199
9 Known Causes of Trouble with GNU CC 205
10 Reporting Bugs 233
11 How To Get Help with GNU CC 243
12 Using GNU CC on VMS 245
Index 251

Table of Contents

GNU GENERAL PUBLIC LICENSE **1**
Preamble 1
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 2
How to Apply These Terms to Your New Programs 8

Contributors to GNU CC **11**

1 Funding Free Software **15**

2 Protect Your Freedom—Fight "Look And Feel" **17**

3 Compile C, C++, or Objective C **21**

4 GNU CC Command Options **23**
4.1 Option Summary 23
4.2 Options Controlling the Kind of Output 28
4.3 Compiling C++ Programs 30
4.4 Options Controlling C Dialect 31
4.5 Options Controlling C++ Dialect 35
4.6 Options to Request or Suppress Warnings 40
4.7 Options for Debugging Your Program or GNU CC 48
4.8 Options That Control Optimization 53
4.9 Options Controlling the Preprocessor 58
4.10 Passing Options to the Assembler 61
4.11 Options for Linking 61
4.12 Options for Directory Search 64
4.13 Specifying Target Machine and Compiler Version 65
4.14 Hardware Models and Configurations 66
4.14.1 M680x0 Options 67
4.14.2 VAX Options 68
4.14.3 SPARC Options 69
4.14.4 Convex Options 72
4.14.5 AMD29K Options 73
4.14.6 ARM Options 74
4.14.7 M88K Options 75
4.14.8 IBM RS/6000 and PowerPC Options 79
4.14.9 IBM RT Options 83
4.14.10 MIPS Options 83

4.14.11 Intel 386 Options 87
4.14.12 HPPA Options 88
4.14.13 Intel 960 Options 90
4.14.14 DEC Alpha Options 91
4.14.15 Clipper Options 92
4.14.16 H8/300 Options 92
4.14.17 Options for System V 92
4.14.18 Zilog Z8000 Option 93
4.14.19 Options for the H8/500 93
4.15 Options for Code Generation Conventions 94
4.16 Environment Variables Affecting GNU CC 98
4.17 Running Protoize 100

5 Installing GNU CC 103

5.1 Configurations Supported by GNU CC 111
5.2 Compilation in a Separate Directory 127
5.3 Building and Installing a Cross-Compiler 127
5.3.1 Steps of Cross-Compilation 128
5.3.2 Configuring a Cross-Compiler 128
5.3.3 Tools and Libraries for a Cross-Compiler 129
5.3.4 'libgcc.a' and Cross-Compilers 130
5.3.5 Cross-Compilers and Header Files 131
5.3.6 Actually Building the Cross-Compiler 132
5.4 Installing GNU CC on the Sun 133
5.5 Installing GNU CC on VMS 133
5.6 collect2 137
5.7 Standard Header File Directories 138

6 Extensions to the C Language Family 141

6.1 Statements and Declarations in Expressions 141
6.2 Locally Declared Labels 142
6.3 Labels as Values 143
6.4 Nested Functions 143
6.5 Constructing Function Calls 146
6.6 Naming an Expression's Type 146
6.7 Referring to a Type with typeof 147
6.8 Generalized Lvalues 148
6.9 Conditionals with Omitted Operands 149
6.10 Double-Word Integers 149
6.11 Complex Numbers 150
6.12 Arrays of Length Zero 151
6.13 Arrays of Variable Length 151
6.14 Macros with Variable Numbers of Arguments 152
6.15 Non-Lvalue Arrays May Have Subscripts 153

6.16 Arithmetic on `void`- and Function-Pointers 154
6.17 Non-Constant Initializers 154
6.18 Constructor Expressions 154
6.19 Labeled Elements in Initializers 155
6.20 Case Ranges 156
6.21 Cast to a Union Type 157
6.22 Declaring Attributes of Functions 157
6.23 Prototypes and Old-Style Function Definitions 160
6.24 Compiling Functions for Interrupt Calls 161
6.25 Dollar Signs in Identifier Names 162
6.26 The Character `ESC` in Constants 162
6.27 Inquiring on Alignment of Types or Variables 162
6.28 Specifying Attributes of Variables 163
6.29 Specifying Attributes of Types 166
6.30 An Inline Function is As Fast As a Macro 169
6.31 Assembler Instructions with C Expression Operands
. . . . 170
6.32 Constraints for `asm` Operands 174
6.32.1 Simple Constraints 174
6.32.2 Multiple Alternative Constraints 177
6.32.3 Constraint Modifier Characters 177
6.32.4 Constraints for Particular Machines 178
6.33 Controlling Names Used in Assembler Code 184
6.34 Variables in Specified Registers 184
6.34.1 Defining Global Register Variables 185
6.34.2 Specifying Registers for Local Variables 186
6.35 Alternate Keywords 187
6.36 Incomplete `enum` Types 187
6.37 Function Names as Strings 188

7 Extensions to the C++ Language 189
7.1 Named Return Values in C++ 189
7.2 Minimum and Maximum Operators in C++ 191
7.3 `goto` and Destructors in GNU C++ 191
7.4 Declarations and Definitions in One Header 191
7.5 Where’s the Template? 193
7.6 Type Abstraction using Signatures 196

8 `gcov`: a Test Coverage Program 199
8.1 Introduction to `gcov` 199
8.2 Invoking gcov 200
8.3 Using `gcov` with GCC Optimization 202

9 Known Causes of Trouble with GNU CC . . . 205
9.1 Actual Bugs We Haven’t Fixed Yet 205
9.2 Installation Problems 205
9.3 Cross-Compiler Problems 211
9.4 Interoperation 211
9.5 Problems Compiling Certain Programs 217
9.6 Incompatibilities of GNU CC 218
9.7 Fixed Header Files 221
9.8 Disappointments and Misunderstandings 222
9.9 Common Misunderstandings with GNU C++ 224
9.9.1 Declare *and* Define Static Members 224
9.9.2 Temporaries May Vanish Before You Expect . . 224
9.10 Caveats of using `protoize` 225
9.11 Certain Changes We Don’t Want to Make 227
9.12 Warning Messages and Error Messages 230

10 Reporting Bugs 233
10.1 Have You Found a Bug? 233
10.2 Where to Report Bugs 234
10.3 How to Report Bugs 235
10.4 Sending Patches for GNU CC 239

11 How To Get Help with GNU CC 243

12 Using GNU CC on VMS 245
12.1 Include Files and VMS 245
12.2 Global Declarations and VMS 246
12.3 Other VMS Issues 248

Index 251

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed

under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it

and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of
what it does.  Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in
the program 'Gnomovision' (which makes passes at compilers)
written by James Hacker.

signature of Ty Coon, 1 April 1989
```

```
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Contributors to GNU CC

In addition to Richard Stallman, several people have written parts of GNU CC.

- The idea of using RTL and some of the optimization ideas came from the program PO written at the University of Arizona by Jack Davidson and Christopher Fraser. See “Register Allocation and Exhaustive Peephole Optimization”, Software Practice and Experience 14 (9), Sept. 1984, 857-866.
- Paul Rubin wrote most of the preprocessor.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the Vax machine description.
- Ted Lemon wrote parts of the RTL reader and printer.
- Jim Wilson implemented loop strength reduction and some other loop optimizations.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Michael Tiemann of Cygnus Support wrote the front end for C++, as well as the support for inline functions and instruction scheduling. Also the descriptions of the National Semiconductor 32000 series cpu, the SPARC cpu and part of the Motorola 88000 cpu.
- Gerald Baumgartner added the signature extension to the C++ front-end.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Randy Smith finished the Sun FPA support.
- Robert Brown implemented the support for Encore 32000 systems.
- David Kashtan of SRI adapted GNU CC to VMS.
- Alex Crain provided changes for the 3b1.
- Greg Satz and Chris Hanson assisted in making GNU CC work on HP-UX for the 9000 series 300.
- William Schelter did most of the work on the Intel 80386 support.
- Christopher Smith did the port for Convex machines.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Alain Lichnewsky ported GNU CC to the Mips cpu.

- Devon Bowen, Dale Wiles and Kevin Zachmann ported GNU CC to the Tahoe.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Gary Miller ported GNU CC to Charles River Data Systems machines.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination.
- Richard Kenner and Michael Tiemann jointly developed reorg.c, the delay slot scheduler.
- Mike Meissner and Tom Wood of Data General finished the port to the Motorola 88000.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- NeXT, Inc. donated the front end that supports the Objective C language.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Mike Meissner at the Open Software Foundation finished the port to the MIPS cpu, including adding ECOFF debug support, and worked on the Intel port for the Intel 80386 cpu.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Torbjorn Granlund of the Swedish Institute of Computer Science implemented multiply-by-constant optimization and better long long support, and improved leaf function register allocation.
- Mike Stump implemented the support for Elxsi 64 bit CPU.
- John Wehle added the machine description for the Western Electric 32000 processor used in several 3b series machines (no relation to the National Semiconductor 32000 processor).
- Holger Teutsch provided the support for the Clipper cpu.
- Kresten Krab Thorup wrote the run time support for the Objective C language.

- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits.
- David Edelsohn contributed the changes to RS/6000 port to make it support the PowerPC and POWER2 architectures.
- Steve Chamberlain wrote the support for the Hitachi SH processor.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.

1 Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU C compiler contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Copyright (C) 1994 Free Software Foundation, Inc.
Verbatim copying and redistribution of this section is permitted without royalty; alteration is not permitted.

2 Protect Your Freedom—Fight “Look And Feel”

This section is a political message from the League for Programming Freedom to the users of GNU CC. We have included it here because the issue of interface copyright is important to the GNU project.

Apple and Lotus have tried to create a new form of legal monopoly: a copyright on a user interface.

An interface is a kind of language—a set of conventions for communication between two entities, human or machine. Until a few years ago, the law seemed clear: interfaces were outside the domain of copyright, so programmers could program freely and implement whatever interface the users demanded. Imitating de-facto standard interfaces, sometimes with improvements, was standard practice in the computer field. These improvements, if accepted by the users, caught on and became the norm; in this way, much progress took place.

Computer users, and most software developers, were happy with this state of affairs. However, large companies such as Apple and Lotus would prefer a different system—one in which they can own interfaces and thereby rid themselves of all serious competitors. They hope that interface copyright will give them, in effect, monopolies on major classes of software.

Other large companies such as IBM and Digital also favor interface monopolies, for the same reason: if languages become property, they expect to own many de-facto standard languages. But Apple and Lotus are the ones who have actually sued. Lotus has won lawsuits against two small companies, which were thus put out of business. Then they sued Borland; this case is now before the court of appeals. Apple’s lawsuit against HP and Microsoft is also being decided by an appeals court. Widespread rumors that Apple had lost the case are untrue; as of July 1994, the final outcome is unknown.

If the monopolists get their way, they will hobble the software field:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to design a different way to start, stop, and steer a car.
- Users will be “locked in” to whichever interface they learn; then they will be prisoners of one supplier, who will charge a monopolistic price.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can afford to sue, they can intimidate

smaller developers with threats even when they don't really have a case.

- Interface improvements will come slower, since incremental evolution through creative partial imitation will no longer occur.

If interface monopolies are accepted, other large companies are waiting to grab theirs:

- Adobe is expected to claim a monopoly on the interfaces of various popular application programs, if Borland's appeal against Lotus fails.
- Open Computing magazine reported a Microsoft vice president as threatening to sue people who copy the interface of Windows.

Users invest a great deal of time and money in learning to use computer interfaces. Far more, in fact, than software developers invest in developing *and even implementing* the interfaces. Whoever can own an interface, has made its users into captives, and misappropriated their investment.

To protect our freedom from monopolies like these, a group of programmers and users have formed a grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose monopolistic practices such as interface copyright and software patents. The League calls for a return to the legal policies of the recent past, in which programmers could program freely. The League is not concerned with free software as an issue, and is not affiliated with the Free Software Foundation.

The League's activities include publicizing the issue, as is being done here, and filing friend-of-the-court briefs on behalf of defendants sued by monopolists. Recently the League filed a friend-of-the-court brief for Borland in its appeal against Lotus.

The League's membership rolls include John McCarthy, inventor of Lisp, Marvin Minsky, founder of the MIT Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

Activist members are especially important, but members who have no time to give are also important. Surveys at major ACM conferences have indicated a vast majority of attendees agree with the League. If just ten percent of the programmers who agree with the League join the League, we will probably triumph.

To join, or for more information, phone (617) 243-4091 or write to:

League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139

You can also send electronic mail to `lpf@uunet.uu.net`.

In addition to joining the League, here are some suggestions from the League for other things you can do to protect your freedom to write programs:

- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Mention that you are a League member in your ‘`.signature`’, and mention the League’s email address for inquiries.
- Ask the companies you consider working for or working with to make statements against software monopolies, and give preference to those that do.
- When employers ask you to sign contracts giving them copyright or patent rights, insist on clauses saying they can use these rights only defensively. Don’t rely on “company policy,” since that can change at any time; don’t rely on an individual executive’s private word, since that person may be replaced. Get a commitment just as binding as the commitment they get from you.
- Write to Congress to explain the importance of this issue.

  House Subcommittee on Intellectual Property
  2137 Rayburn Bldg
  Washington, DC 20515

  Senate Subcommittee on Patents, Trademarks and Copyrights
  United States Senate
  Washington, DC 20510

  (These committees have received lots of mail already; let’s give them even more.)

Democracy means nothing if you don’t use it. Stand up and be counted!

3 Compile C, C++, or Objective C

The C, C++, and Objective C versions of the compiler are integrated; the GNU C compiler can compile programs written in C, C++, or Objective C.

"GCC" is a common shorthand term for the GNU C compiler. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs.

When referring to C++ compilation, it is usual to call the compiler "G++". Since there is only one compiler, it is also accurate to call it "GCC" no matter what the language context; however, the term "G++" is more useful when the emphasis is on compiling C++ programs.

We use the name "GNU CC" to refer to the compilation system as a whole, and more specifically to the language-independent part of the compiler. For example, we refer to the optimization options as affecting the behavior of "GNU CC" or sometimes just "the compiler".

Front ends for other languages, such as Ada 9X, Fortran, Modula-3, and Pascal, are under development. These front-ends, like that for C++, are built in subdirectories of GNU CC and link to it. The result is an integrated compiler that can compile programs written in C, C++, Objective C, or any of the languages for which you have installed front ends.

In this manual, we only discuss the options for the C, Objective-C, and C++ compilers and those of the GNU CC core. Consult the documentation of the other front ends for the options to use when compiling programs written in other languages.

G++ is a *compiler*, not merely a preprocessor. G++ builds object code directly from your C++ program source. There is no intermediate C version of the program. (By contrast, for example, some other implementations use a program that generates a C program from your C++ source.) Avoiding an intermediate C representation of the program means that you get better object code, and better debugging information. The GNU debugger, GDB, works with this information in the object code to give you comprehensive C++ source-level editing capabilities (see section "C and C++" in *Debugging with GDB*).

4 GNU CC Command Options

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. The “overall options” allow you to stop this process at an intermediate stage. For example, the ‘-c’ option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command line options that you can use with GNU CC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

See Section 4.3 “Compiling C++ Programs,” page 30, for a summary of special options for compiling C++ programs.

The `gcc` program accepts options and file names as operands. Many options have multiletter names; therefore multiple single-letter options may *not* be grouped: ‘-dr’ is very different from ‘-d -r’.

You can mix options and other arguments. For the most part, the order you use doesn’t matter. Order does matter when you use several options of the same kind; for example, if you specify ‘-L’ more than once, the directories are searched in the order specified.

Many options have long names starting with ‘-f’ or with ‘-W’—for example, ‘-fforce-mem’, ‘-fstrength-reduce’, ‘-Wformat’ and so on. Most of these have both positive and negative forms; the negative form of ‘-ffoo’ would be ‘-fno-foo’. This manual documents only one of these two forms, whichever one is not the default.

4.1 Option Summary

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

Overall Options

See Section 4.2 “Options Controlling the Kind of Output,” page 28.

```
-c  -S  -E  -o file  -pipe  -v  -x language
```

C Language Options

See Section 4.4 “Options Controlling C Dialect,” page 31.

```
-ansi  -fallow-single-precision -fcond-mismatch  -fno-
asm
-fno-builtin  -fsigned-bitfields  -fsigned-char
-funsigned-bitfields  -funsigned-char
-fwritable-strings -traditional  -traditional-cpp
-trigraphs
```

C++ Language Options

See Section 4.5 “Options Controlling C++ Dialect,” page 35.

```
-fall-virtual  -fdollars-in-identifiers
-felide-constructors -fenum-int-equiv
-fexternal-templates  -fhandle-signatures
-fmemoize-lookups  -fno-default-inline
-fno-gnu-keywords -fnonnull-objects  -foperator-names
-fstrict-prototype -fthis-is-variable -nostdinc++
-traditional  +en
```

Warning Options

See Section 4.6 “Options to Request or Suppress Warnings,” page 40.

```
-fsyntax-only  -pedantic  -pedantic-errors
-w  -W  -Wall  -Waggregate-return
-Wbad-function-cast -Wcast-align -Wcast-qual
-Wchar-subscript  -Wcomment -Wconversion
-Wenum-clash  -Werror  -Wformat -Wid-clash-len
-Wimplicit  -Wimport  -Winline -Wlarger-than-len
-Wmissing-declarations -Wmissing-prototypes
-Wnested-externs -Wno-import  -Woverloaded-virtual
-Wparentheses -Wpointer-arith  -Wredundant-decls
-Wreorder -Wreturn-type -Wshadow -Wstrict-prototypes
-Wswitch -Wsynth  -Wtemplate-debugging -Wtraditional
-Wtrigraphs -Wuninitialized  -Wunused
-Wwrite-strings
```

Debugging Options

See Section 4.7 “Options for Debugging Your Program or GCC,” page 48.

```
-a  -dletters  -fpretend-float
-fprofile-arcs  -ftest-coverage
-g  -glevel -gcoff  -gdwarf  -gdwarf+
-ggdb  -gstabs  -gstabs+  -gxcoff  -gxcoff+
-p  -pg  -print-file-name=library
-print-libgcc-file-name -print-prog-name=program
-print-search-dirs  -save-temps
```

Optimization Options

See Section 4.8 “Options that Control Optimization,” page 53.

```
-fbranch-probabilities
-fcaller-saves  -fcse-follow-jumps
-fcse-skip-blocks -fdelayed-branch
-fexpensive-optimizations -ffast-math  -ffloat-store
-fforce-addr  -fforce-mem -finline-functions
```

```
-fkeep-inline-functions -fno-default-inline
-fno-defer-pop  -fno-function-cse -fno-inline
-fno-peephole  -fomit-frame-pointer
-frerun-cse-after-loop  -fschedule-insns
-fschedule-insns2  -fstrength-reduce  -fthread-jumps
-funroll-all-loops  -funroll-loops
-O  -O0  -O1  -O2  -O3
```

Preprocessor Options

See Section 4.9 “Options Controlling the Preprocessor,” page 58.

```
-Aquestion(answer)  -C  -dD  -dM  -dN
-Dmacro[=defn]  -E  -H
-idirafter dir
-include file  -imacros file
-iprefix file  -iwithprefix dir
-iwithprefixbefore dir  -isystem dir
-M  -MD  -MM  -MMD  -MG  -nostdinc  -P  -trigraphs
-undef  -Umacro  -Wp,option
```

Assembler Option

See Section 4.10 “Passing Options to the Assembler,” page 61.

```
-Wa,option
```

Linker Options

See Section 4.11 “Options for Linking,” page 61.

```
object-file-name
-llibrary  -nostartfiles  -nostdlib
-s  -static  -shared  -symbolic
-Wl,option  -Xlinker option
-u symbol
```

Directory Options

See Section 4.12 “Options for Directory Search,” page 64.

```
-Bprefix  -Idir  -I-  -Ldir
```

Target Options

See Section 4.13 “Target Options,” page 65.

```
-b machine  -V version
```

Machine Dependent Options

See Section 4.14 “Hardware Models and Configurations,” page 66.

M680x0 Options

```
-m68000  -m68020  -m68020-40  -m68030  -m68040
-m68881 -mbitfield  -mc68000  -mc68020  -mfpa
-mnobitfield -mrtd  -mshort  -msoft-float
```

VAX Options

```
-mg  -mgnu  -munix
```

SPARC Options

```
-mapp-regs  -mcypress  -mepilogue  -mflat  -mfpu
-mhard-float -mhard-quad-float  -mno-app-regs
-mno-flat  -mno-fpu -mno-epilogue
-mno-unaligned-doubles -msoft-float
-msoft-quad-float -msparclite  -msupersparc
-munaligned-doubles  -mv8
```

SPARC V9 compilers support the following options in addition to the above:

```
-mmedlow  -mmedany
-mint32  -mint64  -mlong32  -mlong64
-mno-stack-bias  -mstack-bias
```

Convex Options

```
-mc1  -mc2  -mc32  -mc34  -mc38
-margcount  -mnoargcount
-mlong32  -mlong64
-mvolatile-cache  -mvolatile-nocache
```

AMD29K Options

```
-m29000  -m29050  -mbw  -mnbw  -mdw  -mndw
-mlarge  -mnormal  -msmall
-mkernel-registers  -mno-reuse-arg-regs
-mno-stack-check  -mno-storem-bug
-mreuse-arg-regs  -msoft-float  -mstack-check
-mstorem-bug  -muser-registers
```

ARM Options

```
-mapcs -m2 -m3 -m6 -mbsd -mxopen -mno-symrename
```

M88K Options

```
-m88000  -m88100  -m88110  -mbig-pic
-mcheck-zero-division  -mhandle-large-shift
-midentify-revision  -mno-check-zero-division
-mno-ocs-debug-info  -mno-ocs-frame-position
-mno-optimize-arg-area  -mno-serialize-volatile
-mno-underscores  -mocs-debug-info
-mocs-frame-position  -moptimize-arg-area
-mserialize-volatile  -mshort-data-num  -msvr3
-msvr4  -mtrap-large-shift  -muse-div-instruction
-mversion-03.00  -mwarn-passed-structs
```

RS/6000 and PowerPC Options

```
-mcpu=cpu  type
-mpower  -mno-power  -mpower2  -mno-power2
-mpowerpc  -mno-powerpc
-mpowerpc-gpopt  -mno-powerpc-gpopt
-mpowerpc-gfxopt  -mno-powerpc-gfxopt
-mnew-mnemonics  -mno-new-mnemonics
-mfull-toc   -mminimal-toc  -mno-fop-in-toc
-mno-sum-in-toc -msoft-float  -mhard-float
```

```
-mmultiple -mno-multiple -mbit-align
-mno-bit-align -mstrict-align -mno-strict-align
-mrelocatable -mno-relocatable -mtraceback
-mno-traceback
```

RT Options

```
-mcall-lib-mul  -mfp-arg-in-fpregs
-mfp-arg-in-gregs -mfull-fp-blocks
-mhc-struct-return  -min-line-mul
-mminimum-fp-blocks  -mnohc-struct-return
```

MIPS Options

```
-mabicalls  -mcpu=cpu  type
-membedded-data -membedded-pic  -mfp32
-mfp64  -mgas  -mgp32  -mgp64 -mgpopt
-mhalf-pic  -mhard-float  -mint64 -mips1
-mips2 -mips3  -mlong64  -mlong-calls  -mmemcpy
-mmips-as  -mmips-tfile  -mno-abicalls
-mno-embedded-data  -mno-embedded-pic
-mno-gpopt  -mno-long-calls
-mno-memcpy  -mno-mips-tfile  -mno-rnames
-mno-stats -mrnames -msoft-float
-m4650 -msingle-float -mmad
-mstats  -G  num  -nocpp
```

i386 Options

```
-m486  -mieee-fp  -mno-486  -mno-fancy-math-387
-mno-fp-ret-in-387  -msoft-float  -msvr3-shlib
-mno-wide-multiply -mreg-alloc=list
```

HPPA Options

```
-mdisable-fpregs  -mdisable-indexing  -mfast-indirect-
calls
-mgas  -mjump-in-delay -mlong-millicode-calls
-mno-disable-fpregs -mno-disable-indexing
-mno-fast-indirect-calls -mno-gas -mno-jump-in-delay
-mno-millicode-long-calls  -mno-portable-runtime
-mno-soft-float -msoft-float
-mpa-risc-1-0  -mpa-risc-1-1  -mportable-runtime
-mschedule=list
```

Intel 960 Options

```
-mcpu type  -masm-compat  -mclean-linkage
-mcode-align  -mcomplex-addr  -mleaf-procedures
-mic-compat  -mic2.0-compat  -mic3.0-compat
-mintel-asm  -mno-clean-linkage  -mno-code-align
-mno-complex-addr  -mno-leaf-procedures
-mno-old-align  -mno-strict-align
-mno-tail-call -mnumerics  -mold-align
-msoft-float  -mstrict-align -mtail-call
```

DEC Alpha Options
```
-mfp-regs  -mno-fp-regs  -mno-soft-float
-msoft-float
```

Clipper Options
```
-mc300 -mc400
```

H8/300 Options
```
-mrelax  -mh
```

System V Options
```
-Qy  -Qn  -YP,paths  -Ym,dir
```

Z8000 Option
```
-mz8001
```

H8/500 Options
```
-mspace  -mspeed
-mint32  -mcode32  -mdata32
-mtiny  -msmall
-mmedium  -mcompact
-mbig
```

Code Generation Options

See Section 4.15 “Options for Code Generation Conventions,” page 94.

```
-fcall-saved-reg  -fcall-used-reg
-ffixed-reg  -finhibit-size-directive
-fno-common  -fno-ident  -fno-gnu-linker
-fpcc-struct-return  -fpic  -fPIC
-freg-struct-return  -fshared-data  -fshort-enums
-fshort-double  -funaligned-pointers
-funaligned-struct-hack  -fvolatile -fvolatile-global
-fverbose-asm  -fpack-struct -fverbose-asm  +e0  +e1
```

4.2 Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

`file.c` C source code which must be preprocessed.

`file.i` C source code which should not be preprocessed.

`file.ii` C++ source code which should not be preprocessed.

file.m Objective-C source code. Note that you must link with the library '`libobjc.a`' to make an Objective-C program work.

file.h C header file (not to be compiled or linked).

file.cc
file.cxx
file.cpp
file.C C++ source code which must be preprocessed. Note that in '`.cxx`', the last two letters must both be literally '`x`'. Likewise, '`.C`' refers to a literal capital C.

file.s Assembler code.

file.S Assembler code which must be preprocessed.

other An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

You can specify the input language explicitly with the '`-x`' option:

-x *language*
Specify explicitly the *language* for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next '`-x`' option. Possible values for *language* are:

```
c  objective-c  c++
c-header  cpp-output  c++-cpp-output
assembler  assembler-with-cpp
```

-x none Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if '`-x`' has not been used at all).

If you only want some of the stages of compilation, you can use '`-x`' (or filename suffixes) to tell `gcc` where to start, and one of the options '`-c`', '`-S`', or '`-E`' to say where `gcc` is to stop. Note that some combinations (for example, '`-x cpp-output -E`' instruct `gcc` to do nothing at all.

-c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix '`.c`', '`.i`', '`.s`', etc., with '`.o`'.

Unrecognized input files, not requiring compilation or assembly, are ignored.

-S Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix '.c', '.i', etc., with '.s'.

Input files that don't require compilation are ignored.

`-E` Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files which don't require preprocessing are ignored.

`-o file` Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

Since only one output file can be specified, it does not make sense to use '-o' when compiling more than one input file, unless you are producing an executable file as output.

If '-o' is not specified, the default is to put an executable file in 'a.out', the object file for '*source.suffix*' in '*source*.o', its assembler file in '*source*.s', and all preprocessed C source on standard output.

`-v` Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

`-pipe` Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

4.3 Compiling C++ Programs

C++ source files conventionally use one of the suffixes '.C', '.cc', 'cpp', or '.cxx'; preprocessed C++ files use the suffix '.ii'. GNU CC recognizes files with these names and compiles them as C++ programs even if you call the compiler the same way as for compiling C programs (usually with the name gcc).

However, C++ programs often require class libraries as well as a compiler that understands the C++ language—and under some circumstances, you might want to compile programs from standard input, or otherwise without a suffix that flags them as C++ programs. g++ is a program that calls GNU CC with the default language set to C++, and automatically specifies linking against the GNU class library libg++.

[1] On many systems, the script g++ is also installed with the name c++.

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See Section 4.4 "Options Controlling C Dialect," page 31, for explanations of options for languages related to C. See Section 4.5 "Options Controlling C++ Dialect," page 35, for explanations of options that are meaningful only for C++ programs.

4.4 Options Controlling C Dialect

The following options control the dialect of C (or languages derived from C, such as C++ and Objective C) that the compiler accepts:

`-ansi`
: Support all ANSI standard C programs.

  This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, and disallows '`$`' as part of identifiers.

  The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite '`-ansi`'. You would not want to use them in an ANSI C program, of course, but it is useful to put them in header files that might be included in compilations done with '`-ansi`'. Alternate predefined macros such as `__unix__` and `__vax__` are also available, with or without '`-ansi`'.

  The '`-ansi`' option does not cause non-ANSI programs to be rejected gratuitously. For that, '`-pedantic`' is required in addition to '`-ansi`'. See Section 4.6 "Warning Options," page 40.

  The macro `__STRICT_ANSI__` is predefined when the '`-ansi`' option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain

[1] Prior to release 2 of the compiler, there was a separate g++ compiler. That version was based on GNU CC, but not integrated with it. Versions of g++ with a '1.*xx*' version number—for example, g++ version 1.37 or 1.42—are much less reliable than the versions integrated with GCC 2. Moreover, combining G++ '1.*xx*' with a version 2 GCC will simply not work.

macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

The functions `alloca`, `abort`, `exit`, and `_exit` are not builtin functions when '`-ansi`' is used.

`-fno-asm`
: Do not recognize `asm`, `inline` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__asm__`, `__inline__` and `__typeof__` instead. '`-ansi`' implies '`-fno-asm`'.

  In C++, this switch only affects the `typeof` keyword, since `asm` and `inline` are standard keywords. You may want to use the '`-fno-gnu-keywords`' flag instead, as it also disables the other, C++-specific, extension keywords such as `headof`.

`-fno-builtin`
: Don't recognize builtin functions that do not begin with two leading underscores. Currently, the functions affected include `abort`, `abs`, `alloca`, `cos`, `exit`, `fabs`, `ffs`, `labs`, `memcmp`, `memcpy`, `sin`, `sqrt`, `strcmp`, `strcpy`, and `strlen`.

  GCC normally generates special code to handle certain builtin functions more efficiently; for instance, calls to `alloca` may become single instructions that adjust the stack directly, and calls to `memcpy` may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.

  The '`-ansi`' option prevents `alloca` and `ffs` from being builtin functions, since these functions do not have an ANSI standard meaning.

`-trigraphs`
: Support ANSI C trigraphs. You don't want to know about this brain-damage. The '`-ansi`' option implies '`-trigraphs`'.

`-traditional`
: Attempt to support some aspects of traditional C compilers. Specifically:

  - All `extern` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
  - The newer keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized. (You can still use the alternative keywords such as `__typeof__`, `__inline__`, and so on.)

- Comparisons between pointers and integers are always allowed.
- Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.
- Out-of-range floating point literals are not an error.
- Certain constructs which ANSI regards as a single invalid preprocessing number, such as '`0xe-0xd`', are treated as expressions instead.
- String "constants" are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of '`-fwritable-strings`'.)
- All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.
- The character escape sequences '`\x`' and '`\a`' evaluate as the literal characters '`x`' and '`a`' respectively. Without '`-traditional`', '`\x`' is a prefix for the hexadecimal representation of a character, and '`\a`' produces a bell.
- In C++ programs, assignment to `this` is permitted with '`-traditional`'. (The option '`-fthis-is-variable`' also has this effect.)

You may wish to use '`-fno-builtin`' as well as '`-traditional`' if your program uses names that are normally GNU C builtin functions for other purposes of its own.

You cannot use '`-traditional`' if you include any header files that rely on ANSI C features. Some vendors are starting to ship systems with ANSI C header files and you cannot use '`-traditional`' on such systems to compile files that include any system headers.

In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

In preprocessing directive, the '`#`' symbol must appear as the first character of a line.

In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.

The predefined macro `__STDC__` is not defined when you use '`-traditional`', but `__GNUC__` is (since the GNU extensions which `__GNUC__` indicates are not affected by '`-traditional`'). If you need to write header files that work differently depending on whether '`-traditional`' is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers. The predefined macro `__STDC_VERSION__` is also not defined when you use '`-traditional`'. See section "Standard Predefined Macros" in *The C Preprocessor*, for more discussion of these and other predefined macros.

The preprocessor considers a string constant to end at a newline (unless the newline is escaped with '\'). (Without '`-traditional`', string constants can contain the newline character as typed.)

`-traditional-cpp`

Attempt to support some aspects of traditional C preprocessors. This includes the last five items in the table immediately above, but none of the other effects of '`-traditional`'.

`-fcond-mismatch`

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

`-funsigned-char`

Let the type `char` be unsigned, like `unsigned char`.

Each kind of machine has a default for what `char` should be. It is either like `unsigned char` by default or like `signed char` by default.

Ideally, a portable program should always use `signed char` or `unsigned char` when it depends on the signedness of an object. But many programs have been written to use plain `char` and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type `char` is always a distinct type from each of `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

`-fsigned-char`

Let the type `char` be signed, like `signed char`.

Note that this is equivalent to '`-fno-unsigned-char`', which is the negative form of '`-funsigned-char`'. Likewise, the option '`-fno-signed-char`' is equivalent to '`-funsigned-char`'.

`-fsigned-bitfields`
`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`

These options control whether a bitfield is signed or unsigned, when the declaration does not use either `signed` or `unsigned`. By default, such a bitfield is signed, because this is consistent: the basic integer types such as `int` are signed types.

However, when '`-traditional`' is used, bitfields are all unsigned no matter what.

`-fwritable-strings`

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. The option '`-traditional`' also has this effect.

Writing into string constants is a very bad idea; "constants" should be constant.

`-fallow-single-precision`

Do not promote single precision math operations to double precision, even when compiling with '`-traditional`'.

Traditional K&R C promotes all floating point operations to double precision, regardless of the sizes of the operands. On the architecture for which you are compiling, single precision may be faster than double precision. If you must use '`-traditional`', but want to use single precision operations when the operands are single precision, use this option. This option has no effect when compiling with ANSI or GNU C conventions (the default).

4.5 Options Controlling C++ Dialect

This section describes the command-line options that are only meaningful for C++ programs; but you can also use most of the GNU compiler options regardless of what language your program is in. For example, you might compile a file `firstClass.C` like this:

```
g++ -g -felide-constructors -O -c firstClass.C
```

In this example, only '`-felide-constructors`' is an option meant only for C++ programs; you can use the other options with any language supported by GNU CC.

Here is a list of options that are *only* for compiling C++ programs:

`-fno-access-control`
: Turn off all access checking. This switch is mainly useful for working around bugs in the access control code.

`-fall-virtual`
: Treat all possible member functions as virtual, implicitly. All member functions (except for constructor functions and `new` or `delete` member operators) are treated as virtual functions of the class where they appear.

  This does not mean that all calls to these member functions will be made through the internal table of virtual functions. Under some circumstances, the compiler can determine that a call to a given virtual function can be made directly; in these cases the calls are direct in any case.

`-fcheck-new`
: Check that the pointer returned by `operator new` is non-null before attempting to modify the storage allocated. The current Working Paper requires that `operator new` never return a null pointer, so this check is normally unnecessary.

`-fconserve-space`
: Put uninitialized or runtime-initialized global variables into the common segment, as C does. This saves space in the executable at the cost of not diagnosing duplicate definitions. If you compile with this flag and your program mysteriously crashes after `main()` has completed, you may have an object that is being destroyed twice because two definitions were merged.

`-fdollars-in-identifiers`
: Accept '`$`' in identifiers. You can also explicitly prohibit use of '`$`' with the option '`-fno-dollars-in-identifiers`'. (GNU C++ allows '`$`' by default on some target systems but not others.) Traditional C allowed the character '`$`' to form part of identifiers. However, ANSI C and C++ forbid '`$`' in identifiers.

`-fenum-int-equiv`
: Anachronistically permit implicit conversion of `int` to enumeration types. Current C++ allows conversion of `enum` to `int`, but not the other way around.

`-fexternal-templates`
: Cause template instantiations to obey '`#pragma interface`' and '`implementation`'; template instances are emitted or not according to the location of the template definition. See Section 7.5 "Template Instantiation," page 193, for more information.

`-falt-external-templates`
: Similar to -fexternal-templates, but template instances are emitted or not according to the place where they are first instantiated. See Section 7.5 "Template Instantiation," page 193, for more information.

`-fno-gnu-keywords`
: Do not recognize `classof`, `headof`, `signature`, `sigof` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__classof__`, `__headof__`, `__signature__`, `__sigof__`, and `__typeof__` instead. '`-ansi`' implies '`-fno-gnu-keywords`'.

`-fno-implicit-templates`
: Never emit code for templates which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See Section 7.5 "Template Instantiation," page 193, for more information.

`-fhandle-signatures`
: Recognize the `signature` and `sigof` keywords for specifying abstract types. The default ('`-fno-handle-signatures`') is not to recognize them. See Section 7.6 "C++ Signatures," page 196.

`-fhuge-objects`
: Support virtual function calls for objects that exceed the size representable by a '`short int`'. Users should not use this flag by default; if you need to use it, the compiler will tell you so. If you compile any of your code with this flag, you must compile *all* of your code with this flag (including libg++, if you use it).

  This flag is not useful when compiling with -fvtable-thunks.

`-fno-implement-inlines`
: To save space, do not emit out-of-line copies of inline functions controlled by '`#pragma implementation`'. This will cause linker errors if these functions are not inlined everywhere they are called.

`-fmemoize-lookups`
`-fsave-memoized`

Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly.

The first time the compiler must build a call to a member function (or reference to a data member), it must (1) determine whether the class implements member functions of that name; (2) resolve which member function to call (which involves figuring out what sorts of type conversions need to be made); and (3) check the visibility of the member function to the caller. All of this adds up to slower compilation. Normally, the second time a call is made to that member function (or reference to that data member), it must go through the same lengthy process again. This means that code like this:

```
cout << "This " << p << " has " << n << " legs.\n";
```

makes six passes through all three steps. By using a software cache, a “hit” significantly reduces this cost. Unfortunately, using the cache introduces another layer of mechanisms which must be implemented, and so incurs its own overhead. ‘`-fmemoize-lookups`’ enables the software cache.

Because access privileges (visibility) to members and member functions may differ from one function context to the next, G++ may need to flush the cache. With the ‘`-fmemoize-lookups`’ flag, the cache is flushed after every function that is compiled. The ‘`-fsave-memoized`’ flag enables the same software cache, but when the compiler determines that the context of the last function compiled would yield the same access privileges of the next function to compile, it preserves the cache. This is most helpful when defining many member functions for the same class: with the exception of member functions which are friends of other classes, each member function has exactly the same access privileges as every other, and the cache need not be flushed.

The code that implements these flags has rotted; you should probably avoid using them.

`-fstrict-prototype`

Within an ‘`extern "C"`’ linkage specification, treat a function declaration with no arguments, such as ‘`int foo ();`’, as declaring the function to take no arguments. Normally, such a declaration means that the function `foo` can take any combination of arguments, as in C.

'-pedantic' implies '-fstrict-prototype' unless overridden with '-fno-strict-prototype'.

This flag no longer affects declarations with C++ linkage.

-fno-nonnull-objects
: Don't assume that a reference is initialized to refer to a valid object. Although the current C++ Working Paper prohibits null references, some old code may rely on them, and you can use '-fno-nonnull-objects' to turn on checking.

  At the moment, the compiler only does this checking for conversions to virtual base classes.

-foperator-names
: Recognize the operator name keywords and, bitand, bitor, compl, not, or and xor as synonyms for the symbols they refer to. '-ansi' implies '-foperator-names'.

-fthis-is-variable
: Permit assignment to this. The incorporation of user-defined free store management into C++ has made assignment to 'this' an anachronism. Therefore, by default it is invalid to assign to this within a class member function; that is, GNU C++ treats 'this' in a member function of class X as a non-lvalue of type 'X *'. However, for backwards compatibility, you can make it valid with '-fthis-is-variable'.

-fvtable-thunks
: Use 'thunks' to implement the virtual function dispatch table ('vtable'). The traditional (cfront-style) approach to implementing vtables was to store a pointer to the function and two offsets for adjusting the 'this' pointer at the call site. Newer implementations store a single pointer to a 'thunk' function which does any necessary adjustment and then calls the target function.

  This option also enables a heuristic for controlling emission of vtables; if a class has any non-inline virtual functions, the vtable will be emitted in the translation unit containing the first one of those.

-nostdinc++
: Do not search for header files in the standard directories specific to C++, but do still search the other standard directories. (This option is used when building libg++.)

-traditional
: For C++ programs (in addition to the effects that apply to both C and C++), this has the same effect as

'-fthis-is-variable'. See Section 4.4 "Options Controlling C Dialect," page 31.

In addition, these optimization, warning, and code generation options have meanings only for C++ programs:

`-fno-default-inline`
: Do not assume 'inline' for functions defined inside a class scope. See Section 4.8 "Options That Control Optimization," page 53.

`-Wenum-clash`
`-Woverloaded-virtual`
`-Wtemplate-debugging`
: Warnings that apply only to C++ programs. See Section 4.6 "Options to Request or Suppress Warnings," page 40.

`+en`
: Control how virtual function definitions are used, in a fashion compatible with `cfront` 1.x. See Section 4.15 "Options for Code Generation Conventions," page 94.

4.6 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning '-W', for example '-Wimplicit' to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning '-Wno-' to turn off warnings; for example, '-Wno-implicit'. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by GNU CC:

`-fsyntax-only`
: Check the code for syntax errors, but don't do anything beyond that.

`-w`
: Inhibit all warning messages.

`-Wno-import`
: Inhibit warning messages about the use of '#import'.

`-pedantic`
: Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

  Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require

'-ansi'). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

'-pedantic' does not cause warning messages for use of the alternate keywords whose names begin and end with '__'. Pedantic warnings are also disabled in the expression that follows __extension__. However, only system header files should use these escape routes; application programs should avoid them. See Section 6.35 "Alternate Keywords," page 187.

This option is not intended to be *useful*; it exists only to satisfy pedants who would otherwise claim that GNU CC fails to support the ANSI standard.

Some users try to use '-pedantic' to check programs for strict ANSI C conformance. They soon find that it does not do quite what they want: it finds some non-ANSI practices, but not all—only those for which ANSI C *requires* a diagnostic.

A feature to report any failure to conform to ANSI C might be useful in some instances, but would require considerable additional work and would be quite different from '-pedantic'. We recommend, rather, that users take advantage of the extensions of GNU C and disregard the limitations of other compilers. Aside from certain supercomputers and obsolete small machines, there is less and less reason ever to use any other C compiler other than for bootstrapping GNU CC.

-pedantic-errors
: Like '-pedantic', except that errors are produced rather than warnings.

-W
: Print extra warning messages for these events:

- A nonvolatile automatic variable might be changed by a call to longjmp. These warnings as well are possible only in optimizing compilation.

  The compiler sees only the calls to setjmp. It cannot know where longjmp will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because longjmp cannot in fact be called at the place which would cause a problem.

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
  if (a > 0)
    return a;
}
```

- An expression-statement contains no side effects.
- An unsigned value is compared against zero with ‘<’ or ‘<=’.
- A comparison like ‘`x<=y<=z`’ appears; this is equivalent to ‘`(x<=y ? 1 : 0) <= z`’, which is a different interpretation from that of ordinary mathematical notation.
- Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

`-Wimplicit`
: Warn whenever a function or parameter is implicitly declared.

`-Wreturn-type`
: Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

`-Wunused`
: Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used.

To suppress this warning for a local variable or expression, simply cast it to void. This will also work for file-scope variables, but if you want to mark them used at the point of definition, you can use this macro:

```
#define USE(var) \
  static void *const use_##var = (&use_##var, &var, 0)

USE (string);
```

`-Wswitch`
: Warn whenever a `switch` statement has an index of enumeral type and lacks a `case` for one or more of the named codes of that enumeration. (The presence of a `default` label

prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used.

`-Wcomment`
: Warn whenever a comment-start sequence '`/*`' appears in a comment.

`-Wtrigraphs`
: Warn if any trigraphs are encountered (assuming they are enabled).

`-Wformat`
: Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

`-Wchar-subscripts`
: Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

`-Wuninitialized`
: An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify '`-O`', you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  . . .
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only if it is set.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Section 6.22 "Function Attributes," page 157.

`-Wparentheses`
: Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

`-Wenum-clash`
: Warn about conversion between different enumeration types. (C++ only).

`-Wtemplate-debugging`
: When using templates in a C++ program, warn if debugging is not yet fully available (C++ only).

`-Wreorder (C++ only)`
: Warn when the order of member initializers given in the code does not match the order in which they must be executed. For instance:

```
struct A {
  int i;
  int j;
  A(): j (0), i (1) { }
```

```
};
```

Here the compiler will warn that the member initializers for '`i`' and '`j`' will be rearranged to match the declaration order of the members.

`-Wall` All of the above '`-W`' options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining '`-W...`' options are not implied by '`-Wall`' because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

`-Wtraditional`
Warn about certain constructs that behave differently in traditional and ANSI C.

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.
- A function declared external in one block and then used after the end of the block.
- A `switch` statement has an operand of type `long`.

`-Wshadow` Warn whenever a local variable shadows another local variable.

`-Wid-clash-`*`len`*
Warn whenever two distinct identifiers match in the first *`len`* characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

`-Wlarger-than-`*`len`*
Warn whenever an object of larger than *`len`* bytes is defined.

`-Wpointer-arith`
Warn about anything that depends on the "size of" a function type or of `void`. GNU C assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.

`-Wbad-function-cast`
Warn whenever a function call is cast to a non-matching type. For example, warn if `int malloc()` is cast to `anything *`.

`-Wcast-qual`
Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

`-Wcast-align`
Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

`-Wwrite-strings`
Give string constants the type `const char[`*length*`]` so that copying the address of one into a non-`const char *` pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make '`-Wall`' request these warnings.

`-Wconversion`
Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment `x = -1` if `x` is unsigned. But do not warn about explicit casts like `(unsigned) -1`.

`-Waggregate-return`
Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

`-Wstrict-prototypes`
Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

`-Wmissing-prototypes`
Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

`-Wmissing-declarations`
Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a proto-

type. Use this option to detect global functions that are not declared in header files.

`-Wredundant-decls`
Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

`-Wnested-externs`
Warn if an `extern` declaration is encountered within an function.

`-Winline` Warn if a function can not be inlined, and either it was declared as inline, or else the '`-finline-functions`' option was given.

`-Woverloaded-virtual`
Warn when a derived class function declaration may be an error in defining a virtual function (C++ only). In a derived class, the definitions of virtual functions must match the type signature of a virtual function declared in the base class. With this option, the compiler warns when you define a function with the same name as a virtual function, but with a type signature that does not match any declarations from the base class.

`-Wsynth (C++ only)`
Warn when g++'s synthesis behavior does not match that of cfront. For instance:

```
struct A {
  operator int ();
  A& operator = (int);
};

main ()
{
  A a,b;
  a = b;
}
```

In this example, g++ will synthesize a default '`A& operator = (const A&);`', while cfront will use the user-defined '`operator =`'.

`-Werror` Make all warnings into errors.

4.7 Options for Debugging Your Program or GNU CC

GNU CC has various special options that are used for debugging either your program or GCC:

`-g`
: Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

  On most systems that use stabs format, '`-g`' enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use '`-gstabs+`', '`-gstabs`', '`-gxcoff+`', '`-gxcoff`', '`-gdwarf+`', or '`-gdwarf`' (see below).

  Unlike most other C compilers, GNU CC allows you to use '`-g`' with '`-O`'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

  Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

  The following options are useful when GNU CC is generated with the capability for more than one debugging format.

`-ggdb`
: Produce debugging information in the native format (if that is supported), including GDB extensions if at all possible.

`-gstabs`
: Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems. On MIPS, Alpha and System V Release 4 systems this option produces stabs debugging output which is not understood by DBX or SDB. On System V Release 4 systems this option requires the GNU assembler.

`-gstabs+`
: Produce debugging information in stabs format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-gcoff` Produce debugging information in COFF format (if that is supported). This is the format used by SDB on most System V systems prior to System V Release 4.

`-gxcoff` Produce debugging information in XCOFF format (if that is supported). This is the format used by the DBX debugger on IBM RS/6000 systems.

`-gxcoff+` Produce debugging information in XCOFF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program, and may cause assemblers other than the GNU assembler (GAS) to fail with an error.

`-gdwarf` Produce debugging information in DWARF format (if that is supported). This is the format used by SDB on most System V Release 4 systems.

`-gdwarf+` Produce debugging information in DWARF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-g`*`level`*
`-ggdb`*`level`*
`-gstabs`*`level`*
`-gcoff`*`level`*
`-gxcoff`*`level`*
`-gdwarf`*`level`*

Request debugging information and also use *`level`* to specify how much information. The default level is 2.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, but no information about local variables and no line numbers.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use '`-g3`'.

`-p` Generate extra code to write profile information suitable for the analysis program `prof`. You must use this option when compiling the source files you want data about, and you must also use it when linking.

`-pg` Generate extra code to write profile information suitable for the analysis program `gprof`. You must use this option when

compiling the source files you want data about, and you must also use it when linking.

`-a`
: Generate extra code to write profile information for basic blocks, which will record the number of times each basic block is executed, the basic block start address, and the function name containing the basic block. If '`-g`' is used, the line number and filename of the start of the basic block will also be recorded. If not overridden by the machine description, the default action is to append to the text file '`bb.out`'.

  This data could be analyzed by a program like `tcov`. Note, however, that the format of the data is not what `tcov` expects. Eventually GNU `gprof` should be extended to process this data.

`-fprofile-arcs`
: Instrument *arcs* during compilation. For each function of your program, GNU CC creates a program flow graph, then finds a spanning tree for the graph. Only arcs that are not on the spanning tree have to be instrumented: the compiler adds code to count the number of times that these arcs are executed. When an arc is the only exit or only entrance to a block, the instrumentation code can be added to the block; otherwise, a new basic block must be created to hold the instrumentation code.

  Since not every arc in the program must be instrumented, programs compiled with this option run faster than programs compiled with '`-a`', which adds instrumentation code to every basic block in the program. The tradeoff: since `gcov` does not have execution counts for all branches, it must start with the execution counts for the instrumented branches, and then iterate over the program flow graph until the entire graph has been solved. Hence, `gcov` runs a little more slowly than a program which uses information from '`-a`'.

  '`-fprofile-arcs`' also makes it possible to estimate branch probabilities, and to calculate basic block execution counts. In general, basic block execution counts do not give enough information to estimate all branch probabilities. When the compiled program exits, it saves the arc execution counts to a file called '*sourcename*`.da`'. Use the compiler option '`-fbranch-probabilities`' (see Section 4.8 "Options that Control Optimization," page 53) when recompiling, to optimize using estimated branch probabilities.

`-ftest-coverage`

Create data files for the `gcov` code-coverage utility (see Chapter 8 "`gcov`: a GNU CC Test Coverage Program," page 199). The data file names begin with the name of your source file:

sourcename`.bb`

A mapping from basic blocks to line numbers, which `gcov` uses to associate basic block execution counts with line numbers.

sourcename`.bbg`

A list of all arcs in the program flow graph. This allows `gcov` to reconstruct the program flow graph, so that it can compute all basic block and arc execution counts from the information in the *sourcename*`.da` file (this last file is the output from '`-fprofile-arcs`').

`-d`*letters*

Says to make debugging dumps during compilation at times specified by *letters*. This is used for debugging the compiler. The file names for most of the dumps are made by appending a word to the source file name (e.g. '`foo.c.rtl`' or '`foo.c.jump`'). Here are the possible letters for use in *letters*, and their meanings:

'`M`' Dump all macro definitions, at the end of preprocessing, and write no output.

'`N`' Dump all macro names, at the end of preprocessing.

'`D`' Dump all macro definitions, at the end of preprocessing, in addition to normal output.

'`y`' Dump debugging information during parsing, to standard error.

'`r`' Dump after RTL generation, to '*file*`.rtl`'.

'`x`' Just generate RTL for a function instead of compiling it. Usually used with '`r`'.

'`j`' Dump after first jump optimization, to '*file*`.jump`'.

'`s`' Dump after CSE (including the jump optimization that sometimes follows CSE), to '*file*`.cse`'.

'`L`' Dump after loop optimization, to '*file*`.loop`'.

'`t`' Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to '*file*.cse2'.

'`f`' Dump after flow analysis, to '*file*.flow'.

'`c`' Dump after instruction combination, to the file '*file*.combine'.

'`S`' Dump after the first instruction scheduling pass, to '*file*.sched'.

'`l`' Dump after local register allocation, to '*file*.lreg'.

'`g`' Dump after global register allocation, to '*file*.greg'.

'`R`' Dump after the second instruction scheduling pass, to '*file*.sched2'.

'`J`' Dump after last jump optimization, to '*file*.jump2'.

'`d`' Dump after delayed branch scheduling, to '*file*.dbr'.

'`k`' Dump after conversion from registers to stack, to '*file*.stack'.

'`a`' Produce all the dumps listed above.

'`m`' Print statistics on memory usage, at the end of the run, to standard error.

'`p`' Annotate the assembler output with a comment indicating which pattern and alternative was used.

`-fpretend-float`
: When running a cross-compiler, pretend that the target machine uses the same floating point format as the host machine. This causes incorrect output of the actual floating constants, but the actual instruction sequence will probably be the same as GNU CC would make when running on the target machine.

`-save-temps`
: Store the usual "temporary" intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling '`foo.c`' with '`-c -save-temps`' would produce files '`foo.i`' and '`foo.s`', as well as '`foo.o`'.

`-print-file-name=library`
: Print the full absolute name of the library file *library* that would be used when linking—and don't do anything else. With this option, GNU CC does not compile or link anything; it just prints the file name.

`-print-prog-name=program`
: Like '-print-file-name', but searches for a program such as 'cpp'.

`-print-libgcc-file-name`
: Same as '-print-file-name=libgcc.a'.

  This is useful when you use '-nostdlib' but you do want to link with 'libgcc.a'. You can do

```
gcc -nostdlib files... `gcc -print-libgcc-file-name`
```

`-print-search-dirs`
: Print the name of the configured installation directory and a list of program and library directories gcc will search—and don't do anything else.

  This is useful when gcc prints the error message 'installation problem, cannot exec cpp: No such file or directory'. To resolve this you either need to put 'cpp' and the other compiler components where gcc expects to find them, or you can set the environment variable GCC_EXEC_PREFIX to the directory where you installed them. Don't forget the trailing '/'. See Section 4.16 "Environment Variables," page 98.

4.8 Options That Control Optimization

These options control various sorts of optimizations:

`-O`
`-O1`
: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

  Without '-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

  Without '-O', the compiler only allocates variables declared register in registers. The resulting compiled code is a little worse than produced by PCC without '-O'.

With '-O', the compiler tries to reduce code size and execution time.

When you specify '-O', the compiler turns on '-fthread-jumps' and '-fdefer-pop' on all machines. The compiler turns on '-fdelayed-branch' on machines that have delay slots, and '-fomit-frame-pointer' on machines that can support debugging even without a frame pointer. On some machines the compiler also turns on other flags.

-O2
: Optimize even more. GNU CC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify '-O2'. As compared to '-O', this option increases both compilation time and the performance of the generated code.

'-O2' turns on all optional optimizations except for loop unrolling and function inlining. It also turns on frame pointer elimination on machines where doing so does not interfer with debugging.

-O3
: Optimize yet more. '-O3' turns on all optimizations specified by '-O2' and also turns on the 'inline-functions' option.

-O0
: Do not optimize.

If you use multiple '-O' options, with or without level numbers, the last such option is the one that is effective.

Options of the form '-f*flag*' specify machine-independent flags. Most flags have both positive and negative forms; the negative form of '-ffoo' would be '-fno-foo'. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing 'no-' or adding it.

-ffloat-store
: Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory.

This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a double is supposed to have. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '-ffloat-store' for such programs.

-fno-default-inline
: Do not make member functions inline by default merely because they are defined inside the class scope (C++ only). Oth-

erwise, when you specify '-O', member functions defined inside class scope are compiled inline by default; i.e., you don't need to add 'inline' in front of the member function name.

-fno-defer-pop
: Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

-fforce-mem
: Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

-fforce-addr
: Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as '-fforce-mem' may. I am interested in hearing about the difference this makes.

-fomit-frame-pointer
: Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. **It also makes debugging impossible on some machines.**

  On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro FRAME_POINTER_REQUIRED controls whether a target machine supports this flag. See section "Register Usage" in *Using and Porting GCC*.

-fno-inline
: Don't pay attention to the inline keyword. Normally this option is used to keep the compiler from expanding any functions inline. Note that if you are not optimizing, no functions can be expanded inline.

`-finline-functions`
: Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

  If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

`-fkeep-inline-functions`
: Even if all calls to a given function are integrated, and the function is declared `static`, nevertheless output a separate run-time callable version of the function.

`-fno-function-cse`
: Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

  This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

`-ffast-math`
: This option allows GCC to violate some ANSI or IEEE rules and/or specifications in the interest of optimizing code for speed. For example, it allows the compiler to assume arguments to the `sqrt` function are non-negative numbers and that no floating-point values are NaNs.

  This option should never be turned on by any '`-O`' option since it can result in incorrect output for programs which depend on an exact implementation of IEEE or ANSI rules/specifications for math functions.

The following options control specific optimizations. The '`-O2`' option turns on all of these optimizations except '`-funroll-loops`' and '`-funroll-all-loops`'. On most machines, the '`-O`' option turns on the '`-fthread-jumps`' and '`-fdelayed-branch`' options, but specific machines may handle it differently.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

`-fstrength-reduce`
: Perform the optimizations of loop strength reduction and elimination of iteration variables.

`-fthread-jumps`
: Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed

by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

`-fcse-follow-jumps`
: In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an `if` statement with an `else` clause, CSE will follow the jump when the condition tested is false.

`-fcse-skip-blocks`
: This is similar to '`-fcse-follow-jumps`', but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple `if` statement with no else clause, '`-fcse-skip-blocks`' causes CSE to follow the jump around the body of the `if`.

`-frerun-cse-after-loop`
: Re-run common subexpression elimination after loop optimizations has been performed.

`-fexpensive-optimizations`
: Perform a number of minor optimizations that are relatively expensive.

`-fdelayed-branch`
: If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

`-fschedule-insns`
: If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

`-fschedule-insns2`
: Similar to '`-fschedule-insns`', but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

`-fcaller-saves`
: Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save

and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

`-funroll-loops`
: Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. '`-funroll-loop`' implies both '`-fstrength-reduce`' and '`-frerun-cse-after-loop`'.

`-funroll-all-loops`
: Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. '`-funroll-all-loops`' implies '`-fstrength-reduce`' as well as '`-frerun-cse-after-loop`'.

`-fno-peephole`
: Disable any machine-specific peephole optimizations.

`-fbranch-probabilities`
: After running a program compiled with '`-fprofile-arcs`' (see Section 4.7 "Options for Debugging Your Program or `gcc`," page 48), you can compile it a second time using '`-fbranch-probabilities`', to improve optimizations based on guessing the path a branch might take.

4.9 Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the '`-E`' option, nothing is done except preprocessing. Some of these options make sense only together with '`-E`' because they cause the preprocessor output to be unsuitable for actual compilation.

`-include` *`file`*
: Process *`file`* as input before processing the regular input file. In effect, the contents of *`file`* are compiled first. Any '`-D`' and '`-U`' options on the command line are always processed before '`-include` *`file`*', regardless of the order in which they are written. All the '`-include`' and '`-imacros`' options are processed in the order in which they are written.

`-imacros file`
: Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of '`-imacros file`' is to make the macros defined in *file* available for use in the main input.

  Any '`-D`' and '`-U`' options on the command line are always processed before '`-imacros file`', regardless of the order in which they are written. All the '`-include`' and '`-imacros`' options are processed in the order in which they are written.

`-idirafter dir`
: Add the directory *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that '`-I`' adds to).

`-iprefix prefix`
: Specify *prefix* as the prefix for subsequent '`-iwithprefix`' options.

`-iwithprefix dir`
: Add a directory to the second include path. The directory's name is made by concatenating *prefix* and *dir*, where *prefix* was specified previously with '`-iprefix`'. If you have not specified a prefix yet, the directory containing the installed passes of the compiler is used as the default.

`-iwithprefixbefore dir`
: Add a directory to the main include path. The directory's name is made by concatenating *prefix* and *dir*, as in the case of '`-iwithprefix`'.

`-isystem dir`
: Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

`-nostdinc`
: Do not search the standard system directories for header files. Only the directories you have specified with '`-I`' options (and the current directory, if appropriate) are searched. See Section 4.12 "Directory Options," page 64, for information on '`-I`'.

  By using both '`-nostdinc`' and '`-I-`', you can limit the include-file search path to only those directories you specify explicitly.

`-undef` Do not predefine any nonstandard macros. (Including architecture flags).

`-E` Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.

`-C` Tell the preprocessor not to discard comments. Used with the '`-E`' option.

`-P` Tell the preprocessor not to generate '`#line`' directives. Used with the '`-E`' option.

`-M` Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each object file. For each source file, the preprocessor outputs one `make`-rule whose target is the object file name for that source file and whose dependencies are all the `#include` header files it uses. This rule may be a single line or may be continued with '\'-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program.

'`-M`' implies '`-E`'.

Another way to specify output of a `make` rule is by setting the environment variable `DEPENDENCIES_OUTPUT` (see Section 4.16 "Environment Variables," page 98).

`-MM` Like '`-M`' but the output mentions only the user header files included with '`#include "file"`'. System header files included with '`#include <file>`' are omitted.

`-MD` Like '`-M`' but the dependency information is written to a file made by replacing ".c" with ".d" at the end of the input file names. This is in addition to compiling the file as specified—'`-MD`' does not inhibit ordinary compilation the way '`-M`' does.

In Mach, you can use the utility `md` to merge multiple dependency files into a single dependency file suitable for using with the '`make`' command.

`-MMD` Like '`-MD`' except mention only user header files, not system header files.

`-MG` Treat missing header files as generated files and assume they live in the same directory as the source file. If you specify '`-MG`', you must also specify either '`-M`' or '`-MM`'. '`-MG`' is not supported with '`-MD`' or '`-MMD`'.

`-H` Print the name of each header file used, in addition to other normal activities.

`-Aquestion(answer)`
: Assert the answer *answer* for *question*, in case it is tested with a preprocessing conditional such as '`#if #question(answer)`'. '`-A-`' disables the standard assertions that normally describe the target machine.

`-Dmacro`
: Define macro *macro* with the string '`1`' as its definition.

`-Dmacro=defn`
: Define macro *macro* as *defn*. All instances of '`-D`' on the command line are processed before any '`-U`' options.

`-Umacro`
: Undefine macro *macro*. '`-U`' options are evaluated after all '`-D`' options, but before any '`-include`' and '`-imacros`' options.

`-dM`
: Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the '`-E`' option.

`-dD`
: Tell the preprocessing to pass all macro definitions into the output, in their proper sequence in the rest of the output.

`-dN`
: Like '`-dD`' except that the macro arguments and contents are omitted. Only '`#define name`' is included in the output.

`-trigraphs`
: Support ANSI C trigraphs. The '`-ansi`' option also has this effect.

`-Wp,option`
: Pass *option* as an option to the preprocessor. If *option* contains commas, it is split into multiple options at the commas.

4.10 Passing Options to the Assembler

You can pass options to the assembler.

`-Wa,option`
: Pass *option* as an option to the assembler. If *option* contains commas, it is split into multiple options at the commas.

4.11 Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

object-file-name
A file name that does not end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

-c
-S
-E
If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See Section 4.2 "Overall Options," page 28.

-l*library*
Search the library named *library* when linking.

It makes a difference where in the command you write this option; the linker searches processes libraries and object files in the order they are specified. Thus, 'foo.o -lz bar.o' searches library 'z' after file 'foo.o' but before 'bar.o'. If 'bar.o' refers to functions in 'z', those functions may not be loaded.

The linker searches a standard list of directories for the library, which is actually a file named 'lib*library*.a'. The linker then uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with '-L'.

Normally the files found this way are library files—archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an '-l' option and specifying a file name is that '-l' surrounds *library* with 'lib' and '.a' and searches several directories.

-lobjc
You need this special case of the '-l' option in order to link an Objective C program.

-nostartfiles
Do not use the standard system startup files when linking. The standard libraries are used normally.

-nostdlib
Do not use the standard system libraries and startup files when linking. Only the files you specify will be passed to the linker.

One of the standard libraries bypassed by '-nostdlib' is 'libgcc.a', a library of internal subroutines that GNU CC uses to overcome shortcomings of particular machines, or special needs for some languages. (See section "Interfacing to GNU CC Output" in *Porting GNU CC*, for more discussion of 'libgcc.a'.) In most cases, you need 'libgcc.a' even when you want to avoid other standard libraries. In other words, when you specify '-nostdlib' you should usually specify '-lgcc' as well. This ensures that you have no unresolved references to internal GNU CC library subroutines. (For example, '__main', used to ensure C++ constructors will be called; see Section 5.6 "collect2," page 137.)

-s
: Remove all symbol table and relocation information from the executable.

-static
: On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.

-shared
: Produce a shared object which can then be linked with other objects to form an executable. Only a few systems support this option.

-symbolic
: Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option '-Xlinker -z -Xlinker defs'). Only a few systems support this option.

-Xlinker *option*
: Pass *option* as an option to the linker. You can use this to supply system-specific linker options which GNU CC does not know how to recognize.

  If you want to pass an option that takes an argument, you must use '-Xlinker' twice, once for the option and once for the argument. For example, to pass '-assert definitions', you must write '-Xlinker -assert -Xlinker definitions'. It does not work to write '-Xlinker "-assert definitions"', because this passes the entire string as a single argument, which is not what the linker expects.

-Wl,*option*
: Pass *option* as an option to the linker. If *option* contains commas, it is split into multiple options at the commas.

-u *symbol*
: Pretend the symbol *symbol* is undefined, to force linking of library modules to define it. You can use '-u' multiple times

with different symbols to force loading of additional library modules.

4.12 Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

-I*dir*
: Append directory *dir* to the list of directories searched for include files.

-I-
: Any directories you specify with ‘-I’ options before the ‘-I-’ option are searched only for the case of ‘#include "*file*"’; they are not searched for ‘#include <*file*>’.

: If additional directories are specified with ‘-I’ options after the ‘-I-’, these directories are searched for all ‘#include’ directives. (Ordinarily *all* ‘-I’ directories are used this way.)

: In addition, the ‘-I-’ option inhibits the use of the current directory (where the current input file came from) as the first search directory for ‘#include "*file*"’. There is no way to override this effect of ‘-I-’. With ‘-I.’ you can specify searching the directory which was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory.

: ‘-I-’ does not inhibit the use of the standard system directories for header files. Thus, ‘-I-’ and ‘-nostdinc’ are independent.

-L*dir*
: Add directory *dir* to the list of directories to be searched for ‘-l’.

-B*prefix*
: This option specifies where to find the executables, libraries, include files, and data files of the compiler itself.

: The compiler driver program runs one or more of the subprograms ‘cpp’, ‘cc1’, ‘as’ and ‘ld’. It tries *prefix* as a prefix for each program it tries to run, both with and without ‘*machine*/*version*/’ (see Section 4.13 “Target Options,” page 65).

: For each subprogram to be run, the compiler driver first tries the ‘-B’ prefix, if any. If that name is not found, or if ‘-B’ was not specified, the driver tries two standard prefixes, which are ‘/usr/lib/gcc/’ and ‘/usr/local/lib/gcc-lib/’. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your ‘PATH’ environment variable.

'-B' prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into '-L' options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into '-isystem' options for the preprocessor. In this case, the compiler appends 'include' to the prefix.

The run-time support file 'libgcc.a' can also be searched for using the '-B' prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means.

Another way to specify a prefix much like the '-B' prefix is to use the environment variable GCC_EXEC_PREFIX. See Section 4.16 "Environment Variables," page 98.

4.13 Specifying Target Machine and Compiler Version

By default, GNU CC compiles code for the same type of machine that you are using. However, it can also be installed as a cross-compiler, to compile for some other type of machine. In fact, several different configurations of GNU CC, for different target machines, can be installed side by side. Then you specify which one to use with the '-b' option.

In addition, older and newer versions of GNU CC can be installed side by side. One of them (probably the newest) will be the default, but you may sometimes wish to use another.

-b *machine*

The argument *machine* specifies the target machine for compilation. This is useful when you have installed GNU CC as a cross-compiler.

The value to use for *machine* is the same as was specified as the machine type when configuring GNU CC as a cross-compiler. For example, if a cross-compiler was configured with 'configure i386v', meaning to compile for an 80386 running System V, then you would specify '-b i386v' to run that cross compiler.

When you do not specify '-b', it normally means to compile for the same type of machine that you are using.

-V *version*

The argument *version* specifies which version of GNU CC to run. This is useful when multiple versions are installed.

For example, *version* might be '2.0', meaning to run GNU CC version 2.0.

The default version, when you do not specify '-V', is controlled by the way GNU CC is installed. Normally, it will be a version that is recommended for general use.

The '-b' and '-V' options actually work by controlling part of the file name used for the executable files and libraries used for compilation. A given version of GNU CC, for a given target machine, is normally kept in the directory '/usr/local/lib/gcc-lib/*machine*/*version*'.

Thus, sites can customize the effect of '-b' or '-V' either by changing the names of these directories or adding alternate names (or symbolic links). If in directory '/usr/local/lib/gcc-lib/' the file '80386' is a link to the file 'i386v', then '-b 80386' becomes an alias for '-b i386v'.

In one respect, the '-b' or '-V' do not completely change to a different compiler: the top-level driver program gcc that you originally invoked continues to run and invoke the other executables (preprocessor, compiler per se, assembler and linker) that do the real work. However, since no real work is done in the driver program, it usually does not matter that the driver program in use is not the one for the specified target and version.

The only way that the driver program depends on the target machine is in the parsing and handling of special machine-specific options. However, this is controlled by a file which is found, along with the other executables, in the directory for the specified version and target machine. As a result, a single installed driver program adapts to any specified target machine and compiler version.

The driver program executable does control one significant thing, however: the default version and target machine. Therefore, you can install different instances of the driver program, compiled for different targets or versions, under different names.

For example, if the driver for version 2.0 is installed as ogcc and that for version 2.1 is installed as gcc, then the command gcc will use version 2.1 by default, while ogcc will use 2.0 by default. However, you can choose either version with either command with the '-V' option.

4.14 Hardware Models and Configurations

Earlier we discussed the standard option '-b' which chooses among different installed compilers for completely different target machines, such as Vax vs. 68000 vs. 80386.

In addition, each of these target machine types can have its own special options, starting with '-m', to choose among various hardware models

or configurations—for example, 68010 vs 68020, floating coprocessor or none. A single installed version of the compiler can compile for any model or configuration, according to the options specified.

Some configurations of the compiler also support additional special options, usually for compatibility with other compilers on the same platform.

4.14.1 M680x0 Options

These are the '-m' options defined for the 68000 series. The default values for these options depends on which style of 68000 was selected when the compiler was configured; the defaults for the most common choices are given below.

`-m68000`
`-mc68000`
: Generate output for a 68000. This is the default when the compiler is configured for 68000-based systems.

`-m68020`
`-mc68020`
: Generate output for a 68020. This is the default when the compiler is configured for 68020-based systems.

`-m68881`
: Generate output containing 68881 instructions for floating point. This is the default for most 68020 systems unless '-nfp' was specified when the compiler was configured.

`-m68030`
: Generate output for a 68030. This is the default when the compiler is configured for 68030-based systems.

`-m68040`
: Generate output for a 68040. This is the default when the compiler is configured for 68040-based systems.

  This option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040. If your 68040 does not have code to emulate those instructions, use '-m68040'.

`-m68020-40`
: Generate output for a 68040, without using any of the new instructions. This results in code which can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68040.

`-mfpa`
: Generate output containing Sun FPA instructions for floating point.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all

m68k targets. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets 'm68k-*-aout' and 'm68k-*-coff' do provide software floating point support.

-mshort
: Consider type int to be 16 bits wide, like short int.

-mnobitfield
: Do not use the bit-field instructions. The '-m68000' option implies '-mnobitfield'.

-mbitfield
: Do use the bit-field instructions. The '-m68020' option implies '-mbitfield'. This is the default if you use a configuration designed for a 68020.

-mrtd
: Use a different function-calling convention, in which functions that take a fixed number of arguments return with the rtd instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

  This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

  Also, you must provide function prototypes for all functions that take variable numbers of arguments (including printf); otherwise incorrect code will be generated for calls to those functions.

  In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

  The rtd instruction is supported by the 68010 and 68020 processors, but not by the 68000.

4.14.2 VAX Options

These '-m' options are defined for the Vax:

-munix
: Do not output certain jump instructions (aobleq and so on) that the Unix assembler for the Vax cannot handle across long ranges.

-mgnu
: Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

`-mg` Output code for g-format floating point numbers instead of d-format.

4.14.3 SPARC Options

These '`-m`' switches are supported on the SPARC:

`-mno-app-regs`
`-mapp-regs`
: Specify '`-mapp-regs`' to generate output using the global registers 2 through 4, which the SPARC SVR4 ABI reserves for applications. This is the default.

  To be fully SVR4 ABI compliant at the cost of some performance loss, specify '`-mno-app-regs`'. You should compile libraries and system software with this option.

`-mfpu`
`-mhard-float`
: Generate output containing floating point instructions. This is the default.

`-mno-fpu`
`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all SPARC targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets '`sparc-*-aout`' and '`sparclite-*-*`' do provide software floating point support.

  '`-msoft-float`' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile '`libgcc.a`', the library that comes with GNU CC, with '`-msoft-float`' in order for this to work.

`-mhard-quad-float`
: Generate output containing quad-word (long double) floating point instructions.

`-msoft-quad-float`
: Generate output containing library calls for quad-word (long double) floating point instructions. The functions called are those specified in the SPARC ABI. This is the default.

As of this writing, there are no sparc implementations that have hardware support for the quad-word floating point instructions. They all invoke a trap handler for one of these instructions, and then the trap handler emulates the effect of the instruction. Because of the trap handler overhead, this is much slower than calling the ABI library routines. Thus the '`-msoft-quad-float`' option is the default.

`-mno-epilogue`
`-mepilogue`
: With '`-mepilogue`' (the default), the compiler always emits code for function exit at the end of each function. Any function exit in the middle of the function (such as a return statement in C) will generate a jump to the exit code at the end of the function.

  With '`-mno-epilogue`', the compiler tries to emit exit code inline at every function exit.

`-mno-flat`
`-mflat`
: With '`-mflat`', the compiler does not generate save/restore instructions and will use a "flat" or single register window calling convention. This model uses %i7 as the frame pointer and is compatible with the normal register window model. Code from either may be intermixed although debugger support is still incomplete. The local registers and the input registers (0-5) are still treated as "call saved" registers and will be saved on the stack as necessary.

  With '`-mno-flat`' (the default), the compiler emits save/restore instructions (except for leaf functions) and is the normal mode of operation.

`-mno-unaligned-doubles`
`-munaligned-doubles`
: Assume that doubles have 8 byte alignment. This is the default.

  With '`-munaligned-doubles`', GNU CC assumes that doubles have 8 byte alignment only if they are contained in another type, or if they have an absolute address. Otherwise, it assumes they have 4 byte alignment. Specifying this option avoids some rare compatibility problems with code generated by other compilers. It is not the default because it results in a performance loss, especially for floating point code.

`-mv8`
`-msparclite`
: These two options select variations on the SPARC architecture.

By default (unless specifically configured for the Fujitsu SPARClite), GCC generates code for the v7 variant of the SPARC architecture.

'-mv8' will give you SPARC v8 code. The only difference from v7 code is that the compiler emits the integer multiply and integer divide instructions which exist in SPARC v8 but not in SPARC v7.

'-msparclite' will give you SPARClite code. This adds the integer multiply, integer divide step and scan (ffs) instructions which exist in SPARClite but not in SPARC v7.

-mcypress
-msupersparc
: These two options select the processor for which the code is optimised.

With '-mcypress' (the default), the compiler optimizes code for the Cypress CY7C602 chip, as used in the SparcStation/SparcServer 3xx series. This is also apropriate for the older SparcStation 1, 2, IPX etc.

With '-msupersparc' the compiler optimizes code for the SuperSparc cpu, as used in the SparcStation 10, 1000 and 2000 series. This flag also enables use of the full SPARC v8 instruction set.

In a future version of GCC, these options will very likely be renamed to '-mcpu=cypress' and '-mcpu=supersparc'.

These '-m' switches are supported in addition to the above on SPARC V9 processors:

-mmedlow
: Generate code for the Medium/Low code model: assume a 32 bit address space. Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

It is very likely that a future version of GCC will rename this option.

-mmedany
: Generate code for the Medium/Anywhere code model: assume a 32 bit text segment starting at offset 0, and a 32 bit data segment starting anywhere (determined at link time). Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

It is very likely that a future version of GCC will rename this option.

-mint64
: Types long and int are 64 bits.

-mlong32
: Types long and int are 32 bits.

-mlong64
-mint32 Type long is 64 bits, and type int is 32 bits.

-mstack-bias
-mno-stack-bias
With '-mstack-bias', GNU CC assumes that the stack pointer, and frame pointer if present, are offset by -2047 which must be added back when making stack frame references. Otherwise, assume no such offset is present.

4.14.4 Convex Options

These '-m' options are defined for Convex:

-mc1 Generate output for C1. The code will run on any Convex machine. The preprocessor symbol `__convex__c1__` is defined.

-mc2 Generate output for C2. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C2. The preprocessor symbol `__convex_c2__` is defined.

-mc32 Generate output for C32xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C32. The preprocessor symbol `__convex_c32__` is defined.

-mc34 Generate output for C34xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C34. The preprocessor symbol `__convex_c34__` is defined.

-mc38 Generate output for C38xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C38. The preprocessor symbol `__convex_c38__` is defined.

-margcount
Generate code which puts an argument count in the word preceding each argument list. This is compatible with regular CC, and a few programs may need the argument count word. GDB and other source-level debuggers do not need it; this info is in the symbol table.

-mnoargcount
Omit the argument count word. This is the default.

-mvolatile-cache
Allow volatile references to be cached. This is the default.

`-mvolatile-nocache`
: Volatile references bypass the data cache, going all the way to memory. This is only needed for multi-processor code that does not use standard synchronization instructions. Making non-volatile references to volatile locations will not necessarily work.

`-mlong32`
: Type long is 32 bits, the same as type int. This is the default.

`-mlong64`
: Type long is 64 bits, the same as type long long. This option is useless, because no library support exists for it.

4.14.5 AMD29K Options

These '`-m`' options are defined for the AMD Am29000:

`-mdw`
: Generate code that assumes the `DW` bit is set, i.e., that byte and halfword operations are directly supported by the hardware. This is the default.

`-mndw`
: Generate code that assumes the `DW` bit is not set.

`-mbw`
: Generate code that assumes the system supports byte and halfword write operations. This is the default.

`-mnbw`
: Generate code that assumes the systems does not support byte and halfword write operations. '`-mnbw`' implies '`-mndw`'.

`-msmall`
: Use a small memory model that assumes that all function addresses are either within a single 256 KB segment or at an absolute address of less than 256k. This allows the `call` instruction to be used instead of a `const`, `consth`, `calli` sequence.

`-mnormal`
: Use the normal memory model: Generate `call` instructions only when calling functions in the same file and `calli` instructions otherwise. This works if each file occupies less than 256 KB but allows the entire executable to be larger than 256 KB. This is the default.

`-mlarge`
: Always use `calli` instructions. Specify this option if you expect a single file to compile into more than 256 KB of code.

`-m29050`
: Generate code for the Am29050.

`-m29000`
: Generate code for the Am29000. This is the default.

`-mkernel-registers`
: Generate references to registers `gr64-gr95` instead of to registers `gr96-gr127`. This option can be used when compiling

kernel code that wants a set of global registers disjoint from that used by user-mode code.

Note that when this option is used, register names in '-f' flags must use the normal, user-mode, names.

`-muser-registers`
: Use the normal set of global registers, gr96-gr127. This is the default.

`-mstack-check`
`-mno-stack-check`
: Insert (or do not insert) a call to __msp_check after each stack adjustment. This is often used for kernel code.

`-mstorem-bug`
`-mno-storem-bug`
: '-mstorem-bug' handles 29k processors which cannot handle the separation of a mtsrim insn and a storem instruction (most 29000 chips to date, but not the 29050).

`-mno-reuse-arg-regs`
`-mreuse-arg-regs`
: '-mno-reuse-arg-regs' tells the compiler to only use incoming argument registers for copying out arguments. This helps detect calling a function with fewer arguments than it was declared with.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

4.14.6 ARM Options

These '-m' options are defined for Advanced RISC Machines (ARM) architectures:

`-m2`
`-m3`
: These options are identical. Generate code for the ARM2 and ARM3 processors. This option is the default. You should also use this option to generate code for ARM6 processors that are running with a 26-bit program counter.

`-m6`
: Generate code for the ARM6 processor when running with a 32-bit program counter.

`-mapcs` Generate a stack frame that is compliant with the ARM Proceedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code.

`-mbsd` This option only applies to RISC iX. Emulate the native BSD-mode compiler. This is the default if '`-ansi`' is not specified.

`-mxopen` This option only applies to RISC iX. Emulate the native X/Open-mode compiler.

`-mno-symrename`
This option only applies to RISC iX. Do not run the assembler post-processor, '`symrename`', after code has been assembled. Normally it is necessary to modify some of the standard symbols in preparation for linking with the RISC iX C library; this option suppresses this pass. The post-processor is never run when the compiler is built for cross-compilation.

4.14.7 M88K Options

These '`-m`' options are defined for Motorola 88k architectures:

`-m88000` Generate code that works well on both the m88100 and the m88110.

`-m88100` Generate code that works best for the m88100, but that also runs on the m88110.

`-m88110` Generate code that works best for the m88110, and may not run on the m88100.

`-mbig-pic`
Obsolete option to be removed from the next revision. Use '`-fPIC`'.

`-midentify-revision`
Include an `ident` directive in the assembler output recording the source file name, compiler name and version, timestamp, and compilation flags used.

`-mno-underscores`
In assembler output, emit symbol names without adding an underscore character at the beginning of each name. The default is to use an underscore as prefix on each name.

`-mocs-debug-info`
`-mno-ocs-debug-info`
Include (or omit) additional debugging information (about registers used in each stack frame) as specified in the 88open

Object Compatibility Standard, "OCS". This extra information allows debugging of code that has had the frame pointer eliminated. The default for DG/UX, SVr4, and Delta 88 SVr3.2 is to include this information; other 88k configurations omit this information by default.

`-mocs-frame-position`
: When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the canonical frame address, which is the stack pointer (register 31) on entry to the function. The DG/UX, SVr4, Delta88 SVr3.2, and BCS configurations use '`-mocs-frame-position`'; other 88k configurations have the default '`-mno-ocs-frame-position`'.

`-mno-ocs-frame-position`
: When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the frame pointer register (register 30). When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the -g switch.

`-moptimize-arg-area`
`-mno-optimize-arg-area`
: Control how function arguments are stored in stack frames. '`-moptimize-arg-area`' saves space by optimizing them, but this conflicts with the 88open specifications. The opposite alternative, '`-mno-optimize-arg-area`', agrees with 88open standards. By default GNU CC does not optimize the argument area.

`-mshort-data-`*num*
: Generate smaller data references by making them relative to `r0`, which allows loading a value using a single instruction (rather than the usual two). You control which data references are affected by specifying *num* with this option. For example, if you specify '`-mshort-data-512`', then the data references affected are those involving displacements of less than 512 bytes. '`-mshort-data-`*num*' is not effective for *num* greater than 64k.

`-mserialize-volatile`
`-mno-serialize-volatile`
: Do, or don't, generate code to guarantee sequential consistency of volatile memory references. By default, consistency is guaranteed.

The order of memory references made by the MC88110 processor does not always match the order of the instructions

requesting those references. In particular, a load instruction may execute before a preceding store instruction. Such reordering violates sequential consistency of volatile memory references, when there are multiple processors. When consistency must be guaranteed, GNU C generates special instructions, as needed, to force execution in the proper order.

The MC88100 processor does not reorder memory references and so always provides sequential consistency. However, by default, GNU C generates the special instructions to guarantee consistency even when you use '`-m88100`', so that the code may be run on an MC88110 processor. If you intend to run your code only on the MC88100 processor, you may use '`-mno-serialize-volatile`'.

The extra code generated to guarantee consistency may affect the performance of your application. If you know that you can safely forgo this guarantee, you may use '`-mno-serialize-volatile`'.

`-msvr4`
`-msvr3`
Turn on ('`-msvr4`') or off ('`-msvr3`') compiler extensions related to System V release 4 (SVr4). This controls the following:

1. Which variant of the assembler syntax to emit.
2. '`-msvr4`' makes the C preprocessor recognize '`#pragma weak`' that is used on System V release 4.
3. '`-msvr4`' makes GNU CC issue additional declaration directives used in SVr4.

'`-msvr4`' is the default for the m88k-motorola-sysv4 and m88k-dg-dgux m88k configurations. '`-msvr3`' is the default for all other m88k configurations.

`-mversion-03.00`
This option is obsolete, and is ignored.

`-mno-check-zero-division`
`-mcheck-zero-division`
Do, or don't, generate code to guarantee that integer division by zero will be detected. By default, detection is guaranteed.

Some models of the MC88100 processor fail to trap upon integer division by zero under certain conditions. By default, when compiling code that might be run on such a processor, GNU C generates code that explicitly checks for zero-valued divisors and traps with exception number 503 when one is

detected. Use of mno-check-zero-division suppresses such checking for code generated to run on an MC88100 processor.

GNU C assumes that the MC88110 processor correctly detects all instances of integer division by zero. When '-m88110' is specified, both '-mcheck-zero-division' and '-mno-check-zero-division' are ignored, and no explicit checks for zero-valued divisors are generated.

`-muse-div-instruction`

Use the div instruction for signed integer division on the MC88100 processor. By default, the div instruction is not used.

On the MC88100 processor the signed integer division instruction div) traps to the operating system on a negative operand. The operating system transparently completes the operation, but at a large cost in execution time. By default, when compiling code that might be run on an MC88100 processor, GNU C emulates signed integer division using the unsigned integer division instruction divu), thereby avoiding the large penalty of a trap to the operating system. Such emulation has its own, smaller, execution cost in both time and space. To the extent that your code's important signed integer division operations are performed on two nonnegative operands, it may be desirable to use the div instruction directly.

On the MC88110 processor the div instruction (also known as the divs instruction) processes negative operands without trapping to the operating system. When '-m88110' is specified, '-muse-div-instruction' is ignored, and the div instruction is used for signed integer division.

Note that the result of dividing INT_MIN by -1 is undefined. In particular, the behavior of such a division with and without '-muse-div-instruction' may differ.

`-mtrap-large-shift`
`-mhandle-large-shift`

Include code to detect bit-shifts of more than 31 bits; respectively, trap such shifts or emit code to handle them properly. By default GNU CC makes no special provision for large bit shifts.

`-mwarn-passed-structs`

Warn when a function passes a struct as an argument or result. Structure-passing conventions have changed during the evolution of the C language, and are often the source of

portability problems. By default, GNU CC issues no such warning.

4.14.8 IBM RS/6000 and PowerPC Options

These '-m' options are defined for the IBM RS/6000 and PowerPC:

-mpower
-mno-power
-mpower2
-mno-power2
-mpowerpc
-mno-powerpc
-mpowerpc-gpopt
-mno-powerpc-gpopt
-mpowerpc-gfxopt
-mno-powerpc-gfxopt

GNU CC supports two related instruction set architectures for the RS/6000 and PowerPC. The *POWER* instruction set are those instructions supported by the 'rios' chip set used in the original RS/6000 systems and the *PowerPC* instruction set is the architecture of the Motorola MPC6xx microprocessors. The PowerPC architecture defines 64-bit instructions, but they are not supported by any current processors.

Neither architecture is a subset of the other. However there is a large common subset of instructions supported by both. An MQ register is included in processors supporting the POWER architecture.

You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GNU CC. Specifying the '-mcpu=*cpu_type*' overrides the specification of these options. We recommend you use that option rather than these.

The '-mpower' option allows GNU CC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying '-mpower2' implies '-power' and also allows GNU CC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

The '-mpowerpc' option allows GNU CC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture. Specifying '-mpowerpc-gpopt' implies '-mpowerpc' and also allows GNU CC to use the op-

tional PowerPC architecture instructions in the General Purpose group, including floating-point square root. Specifying '-mpowerpc-gfxopt' implies '-mpowerpc' and also allows GNU CC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

If you specify both '-mno-power' and '-mno-powerpc', GNU CC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. Specifying both '-mpower' and '-mpowerpc' permits GNU CC to use any instruction from either architecture and to allow use of the MQ register; specify this for the Motorola MPC601.

`-mnew-mnemonics`
`-mold-mnemonics`
Select which mnemonics to use in the generated assembler code. '-mnew-mnemonics' requests output that uses the assembler mnemonics defined for the PowerPC architecture, while '-mold-mnemonics' requests the assembler mnemonics defined for the POWER architecture. Instructions defined in only one architecture have only one mnemonic; GNU CC uses that mnemonic irrespective of which of thse options is specified.

PowerPC assemblers support both the old and new mnemonics, as will later POWER assemblers. Current POWER assemblers only support the old mnemonics. Specify '-mnew-mnemonics' if you have an assembler that supports them, otherwise specify '-mold-mnemonics'.

The default value of these options depends on how GNU CC was configured. Specifing '-mcpu=*cpu_type*' sometimes overrides the value of these option. Unless you are building a cross-compiler, you should normally not specify either '-mnew-mnemonics' or '-mold-mnemonics', but should instead accept the default.

`-mcpu=`*cpu_type*
Set architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type *cpu_type*. By default, *cpu_type* is the target system defined when GNU CC was configured. Supported values for *cpu_type* are 'rios1', 'rios2', 'rsc', '601', '603', '604', 'power', 'powerpc', '403', and 'common'. '-mcpu=power' and '-mcpu=powerpc' specify generic POWER and pure PowerPC (i.e., not MPC601) architecture machine types, with an ap-

propriate, generic processor model assumed for scheduling purposes.

Specifying '-mcpu=rios1', '-mcpu=rios2', '-mcpu=rsc', or '-mcpu=power' enables the '-mpower' option and disables the '-mpowerpc' option; '-mcpu=601' enables both the '-mpower' and '-mpowerpc' options; '-mcpu=603', '-mcpu=604', '-mcpu=403', and '-mcpu=powerpc' enable the '-mpowerpc' option and disable the '-mpower' option; '-mcpu=common' disables both the '-mpower' and '-mpowerpc' options.

To generate code that will operate on all members of the RS/6000 and PowerPC families, specify '-mcpu=common'. In that case, GNU CC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. GNU CC assumes a generic processor model for scheduling purposes.

Specifying '-mcpu=rios1', '-mcpu=rios2', '-mcpu=rsc', or '-mcpu=power' also disables the 'new-mnemonics' option. Specifying '-mcpu=601', '-mcpu=603', '-mcpu=604', '403', or '-mcpu=powerpc' also enables the 'new-mnemonics' option.

`-mfull-toc`
`-mno-fp-in-toc`
`-mno-sum-in-toc`
`-mminimal-toc`

Modify generation of the TOC (Table Of Contents), which is created for every executable file. The '-mfull-toc' option is selected by default. In that case, GNU CC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GNU CC will also place floating-point constants in the TOC. However, only 16,384 entries are available in the TOC.

If you receive a linker error message that saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with the '-mno-fp-in-toc' and '-mno-sum-in-toc' options. '-mno-fp-in-toc' prevents GNU CC from putting floating-point constants in the TOC and '-mno-sum-in-toc' forces GNU CC to generate code to calculate the sum of an address and a constant at run-time instead of putting that sum into the TOC. You may specify one or both of these options. Each causes GNU CC to produce very slightly slower and larger code at the expense of conserving TOC space.

If you still run out of space in the TOC even when you specify both of these options, specify '`-mminimal-toc`' instead. This option causes GNU CC to make only one TOC entry for every file. When you specify this option, GNU CC will produce code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently executed code.

`-msoft-float`
`-mhard-float`
: Generate code that does not use (uses) the floating-point register set. Software floating point emulation is provided if you use the '`-msoft-float`' option.

`-mmultiple`
`-mno-multiple`
: Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC system.

`-mno-bit-align`
`-mbit-align`
: On embedded PowerPC systems do not (do) force structures and unions that contain bit fields to be aligned to the base type of the bit field.

For example, by default a structure containing nothing but 8 `unsigned` bitfields of length 1 would be aligned to a 4 byte boundary and have a size of 4 bytes. By using '`-mno-bit-align`', the structure would be aligned to a 1 byte boundary and be one byte in size.

`-mno-strict-align`
`-mstrict-align`
: On embedded PowerPC systems do not (do) assume that unaligned memory references will be handled by the system.

`-mrelocatable`
`-mno-relocatable`
: On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime.

`-mno-traceback`
`-mtraceback`
: On embedded PowerPC systems do not (do) generate a traceback tag before the start of the function. This tag can be used by the debugger to identify where the start of a function is.

4.14.9 IBM RT Options

These '`-m`' options are defined for the IBM RT PC:

`-min-line-mul`
: Use an in-line code sequence for integer multiplies. This is the default.

`-mcall-lib-mul`
: Call `lmul$$` for integer multiples.

`-mfull-fp-blocks`
: Generate full-size floating point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

`-mminimum-fp-blocks`
: Do not include extra scratch space in floating point data blocks. This results in smaller code, but slower execution, since scratch space must be allocated dynamically.

`-mfp-arg-in-fpregs`
: Use a calling sequence incompatible with the IBM calling convention in which floating point arguments are passed in floating point registers. Note that `varargs.h` and `stdargs.h` will not work with floating point operands if this option is specified.

`-mfp-arg-in-gregs`
: Use the normal calling convention for floating point arguments. This is the default.

`-mhc-struct-return`
: Return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC (hc) compiler. Use the option '`-fpcc-struct-return`' for compatibility with the Portable C Compiler (pcc).

`-mnohc-struct-return`
: Return some structures of more than one word in registers, when convenient. This is the default. For compatibility with the IBM-supplied compilers, use the option '`-fpcc-struct-return`' or the option '`-mhc-struct-return`'.

4.14.10 MIPS Options

These '`-m`' options are defined for the MIPS family of computers:

`-mcpu=cpu type`
: Assume the defaults for the machine type *cpu type* when scheduling instructions. The choices for *cpu type* are 'r2000', 'r3000', 'r4000', 'r4400', 'r4600', and 'r6000'. While picking a specific *cpu type* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (instruction set architecture) without the '-mips2' or '-mips3' switches being used.

`-mips1`
: Issue instructions from level 1 of the MIPS ISA. This is the default. 'r3000' is the default *cpu type* at this ISA level.

`-mips2`
: Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). 'r6000' is the default *cpu type* at this ISA level.

`-mips3`
: Issue instructions from level 3 of the MIPS ISA (64 bit instructions). 'r4000' is the default *cpu type* at this ISA level. This option does not change the sizes of any of the C data types.

`-mfp32`
: Assume that 32 32-bit floating point registers are available. This is the default.

`-mfp64`
: Assume that 32 64-bit floating point registers are available. This is the default when the '-mips3' option is used.

`-mgp32`
: Assume that 32 32-bit general purpose registers are available. This is the default.

`-mgp64`
: Assume that 32 64-bit general purpose registers are available. This is the default when the '-mips3' option is used.

`-mint64`
: Types long, int, and pointer are 64 bits. This works only if '-mips3' is also specified.

`-mlong64`
: Types long and pointer are 64 bits, and type int is 32 bits. This works only if '-mips3' is also specified.

`-mmips-as`
: Generate code for the MIPS assembler, and invoke 'mips-tfile' to add normal debug information. This is the default for all platforms except for the OSF/1 reference platform, using the OSF/rose object format. If the either of the '-gstabs' or '-gstabs+' switches are used, the 'mips-tfile' program will encapsulate the stabs within MIPS ECOFF.

`-mgas`
: Generate code for the GNU assembler. This is the default on the OSF/1 reference platform, using the OSF/rose object format.

`-mrnames`
`-mno-rnames`
The '`-mrnames`' switch says to output code using the MIPS software names for the registers, instead of the hardware names (ie, *a0* instead of *$4*). The only known assembler that supports this option is the Algorithmics assembler.

`-mgpopt`
`-mno-gpopt`
The '`-mgpopt`' switch says to write all of the data declarations before the instructions in the text section, this allows the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default if optimization is selected.

`-mstats`
`-mno-stats`
For each non-inline function processed, the '`-mstats`' switch causes the compiler to emit one line to the standard error file to print statistics about the program (number of registers saved, stack size, etc.).

`-mmemcpy`
`-mno-memcpy`
The '`-mmemcpy`' switch makes all block moves call the appropriate string function ('`memcpy`' or '`bcopy`') instead of possibly generating inline code.

`-mmips-tfile`
`-mno-mips-tfile`
The '`-mno-mips-tfile`' switch causes the compiler not postprocess the object file with the '`mips-tfile`' program, after the MIPS assembler has generated it to add debug support. If '`mips-tfile`' is not run, then no local variables will be available to the debugger. In addition, '`stage2`' and '`stage3`' objects will have the temporary file names passed to the assembler embedded in the object file, which means the objects will not compare the same. The '`-mno-mips-tfile`' switch should only be used when there are bugs in the '`mips-tfile`' program that prevents compilation.

`-msoft-float`
Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mhard-float`
: Generate output containing floating point instructions. This is the default if you use the unmodified sources.

`-mabicalls`
`-mno-abicalls`
: Emit (or do not emit) the pseudo operations '`.abicalls`', '`.cpload`', and '`.cprestore`' that some System V.4 ports use for position independent code.

`-mlong-calls`
`-mno-long-calls`
: Do all calls with the '`JALR`' instruction, which requires loading up a function's address into a register before the call. You need to use this switch, if you call outside of the current 512 megabyte segment to functions that are not through pointers.

`-mhalf-pic`
`-mno-half-pic`
: Put pointers to extern references into the data section and load them up, rather than put the references in the text section.

`-membedded-pic`
`-mno-embedded-pic`
: Generate PIC code suitable for some embedded systems. All calls are made using PC relative address, and all data is addressed using the $gp register. This requires GNU as and GNU ld which do most of the work.

`-membedded-data`
`-mno-embedded-data`
: Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

`-msingle-float`
`-mdouble-float`
: The '`-msingle-float`' switch tells gcc to assume that the floating point coprocessor only supports single precision operations, as on the '`r4650`' chip. The '`-mdouble-float`' switch permits gcc to use double precision operations. This is the default.

`-mmad`

`-mno-mad` Permit use of the '`mad`', '`madu`' and '`mul`' instructions, as on the '`r4650`' chip.

`-m4650` Turns on '`-msingle-float`', '`-mmad`', and, at least for now, '`-mcpu=r4650`'.

`-G` *num* Put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (*gp* or *$28*), instead of the normal two words used. By default, *num* is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The '`-G` *num*' switch is also passed to the assembler and linker. All modules should be compiled with the same '`-G` *num*' value.

`-nocpp` Tell the MIPS assembler to not run it's preprocessor over user assembler files (with a '`.s`' suffix) when assembling them.

4.14.11 Intel 386 Options

These '`-m`' options are defined for the i386 family of computers:

`-m486`
`-mno-486` Control whether or not code is optimized for a 486 instead of an 386. Code generated for an 486 will run on a 386 and vice versa.

`-mieee-fp`
`-m-no-ieee-fp`
Control whether or not the compiler uses IEEE floating point comparisons. These handle correctly the case where the result of a comparison is unordered.

`-msoft-float`
Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

On machines where a function returns floating point results in the 80387 register stack, some floating point opcodes may be emitted even if '`-msoft-float`' is used.

`-mno-fp-ret-in-387`
Do not use the FPU registers for return values of functions.

The usual calling convention has functions return values of types `float` and `double` in an FPU register, even if there is no FPU. The idea is that the operating system should emulate an FPU.

The option '`-mno-fp-ret-in-387`' causes such values to be returned in ordinary CPU registers instead.

`-mno-fancy-math-387`
: Some 387 emulators do not support the `sin`, `cos` and `sqrt` instructions for the 387. Specify this option to avoid generating those instructions. This option is the default on FreeBSD. As of revision 2.6.1, these instructions are not generated unless you also use the '`-ffast-math`' switch.

`-msvr3-shlib`
`-mno-svr3-shlib`
: Control whether GNU CC places uninitialized locals into `bss` or `data`. '`-msvr3-shlib`' places these locals into `bss`. These options are meaningful only on System V Release 3.

`-mno-wide-multiply`
`-mwide-multiply`
: Control whether GNU CC uses the `mul` and `imul` that produce 64 bit results in `eax:edx` from 32 bit operands to do `long long` multiplies and 32-bit division by constants.

`-mreg-alloc=`*regs*
: Control the default allocation order of integer registers. The string *regs* is a series of letters specifing a register. The supported letters are: `a` allocate EAX; `b` allocate EBX; `c` allocate ECX; `d` allocate EDX; `S` allocate ESI; `D` allocate EDI; `B` allocate EBP.

4.14.12 HPPA Options

These '`-m`' options are defined for the HPPA family of computers:

`-mpa-risc-1-0`
: Generate code for a PA 1.0 processor.

`-mpa-risc-1-1`
: Generate code for a PA 1.1 processor.

`-mjump-in-delay`
: Fill delay slots of function calls with unconditional jump instructions by modifying the return pointer for the function call to be the target of the conditional jump.

`-mmillicode-long-calls`
: Generate code which assumes millicode routines can not be reached by the standard millicode call sequence, linker-generated long-calls, or linker-modified millicode calls. In practice this should only be needed for dynamicly linked executables with extremely large SHLIB_INFO sections.

`-mdisable-fpregs`
: Prevent floating point registers from being used in any manner. This is necessary for compiling kernels which perform lazy context switching of floating point registers. If you use this option and attempt to perform floating point operations, the compiler will abort.

`-mdisable-indexing`
: Prevent the compiler from using indexing address modes. This avoids some rather obscure problems when compiling MIG generated code under MACH.

`-mfast-indirect-calls`
: Generate code which performs faster indirect calls. Such code is suitable for kernels and for static linking. The fast indirect call code will fail miserably if it's part of a dynamically linked executable and in the presense of nested functions.

`-mportable-runtime`
: Use the portable calling conventions proposed by HP for ELF systems.

`-mgas`
: Enable the use of assembler directives only GAS understands.

`-mschedule=`*cpu type*
: Schedule code according to the constraints for the machine type *cpu type*. The choices for *cpu type* are '`700`' for 7*n*0 machines, '`7100`' for 7*n*5 machines, and '`7100`' for 7*n*2 machines. '`700`' is the default for *cpu type*.

  Note the '`7100LC`' scheduling information is incomplete and using '`7100LC`' often leads to bad schedules. For now it's probably best to use '`7100`' instead of '`7100LC`' for the 7*n*2 machines.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all HPPA targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to pro-

vide suitable library functions for cross-compilation. The embedded target '`hppa1.1-*-pro`' does provide software floating point support.

'`-msoft-float`' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile '`libgcc.a`', the library that comes with GNU CC, with '`-msoft-float`' in order for this to work.

4.14.13 Intel 960 Options

These '`-m`' options are defined for the Intel 960 implementations:

`-m`*`cpu type`*
: Assume the defaults for the machine type *`cpu type`* for some of the other options, including instruction scheduling, floating point support, and addressing modes. The choices for *`cpu type`* are '`ka`', '`kb`', '`mc`', '`ca`', '`cf`', '`sa`', and '`sb`'. The default is '`kb`'.

`-mnumerics`
`-msoft-float`
: The '`-mnumerics`' option indicates that the processor does support floating-point instructions. The '`-msoft-float`' option indicates that floating-point support should not be assumed.

`-mleaf-procedures`
`-mno-leaf-procedures`
: Do (or do not) attempt to alter leaf procedures to be callable with the `bal` instruction as well as `call`. This will result in more efficient code for explicit calls when the `bal` instruction can be substituted by the assembler or linker, but less efficient code in other cases, such as calls via function pointers, or using a linker that doesn't support this optimization.

`-mtail-call`
`-mno-tail-call`
: Do (or do not) make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete. The default is '`-mno-tail-call`'.

`-mcomplex-addr`
`-mno-complex-addr`
Assume (or do not assume) that the use of a complex addressing mode is a win on this implementation of the i960. Complex addressing modes may not be worthwhile on the K-series, but they definitely are on the C-series. The default is currently '`-mcomplex-addr`' for all processors except the CB and CC.

`-mcode-align`
`-mno-code-align`
Align code to 8-byte boundaries for faster fetching (or don't bother). Currently turned on by default for C-series implementations only.

`-mic-compat`
`-mic2.0-compat`
`-mic3.0-compat`
Enable compatibility with iC960 v2.0 or v3.0.

`-masm-compat`
`-mintel-asm`
Enable compatibility with the iC960 assembler.

`-mstrict-align`
`-mno-strict-align`
Do not permit (do permit) unaligned accesses.

`-mold-align`
Enable structure-alignment compatibility with Intel's gcc release version 1.3 (based on gcc 1.37). Currently this is buggy in that '`#pragma align 1`' is always assumed as well, and cannot be turned off.

4.14.14 DEC Alpha Options

These '`-m`' options are defined for the DEC Alpha implementations:

`-mno-soft-float`
`-msoft-float`
Use (do not use) the hardware floating-point instructions for floating-point operations. When `-msoft-float` is specified, functions in '`libgcc1.c`' will be used to perform floating-point operations. Unless they are replaced by routines that emulate the floating-point operations, or compiled in such a way as to call such emulations routines, these routines will issue floating-point operations. If you are compiling for

an Alpha without floating-point operations, you must ensure that the library is built so as not to call them.

Note that Alpha implementations without floating-point operations are required to have floating-point registers.

`-mfp-reg`
`-mno-fp-regs`
Generate code that uses (does not use) the floating-point register set. `-mno-fp-regs` implies `-msoft-float`. If the floating-point register set is not used, floating point operands are passed in integer registers as if they were integers and floating-point results are passed in $0 instead of $f0. This is a non-standard calling sequence, so any function with a floating-point argument or return value called by code compiled with `-mno-fp-regs` must also be compiled with that option.

A typical use of this option is building a kernel that does not use, and hence need not save and restore, any floating-point registers.

4.14.15 Clipper Options

These '`-m`' options are defined for the Clipper implementations:

`-mc300` Produce code for a C300 Clipper processor. This is the default.

`-mc400` Produce code for a C400 Clipper processor i.e. use floting point registers f8..f15.

4.14.16 H8/300 Options

These '`-m`' options are defined for the H8/300 implementations:

`-mrelax` Shorten some address references at link time, when possible; uses the linker option '`-relax`'. See section "`ld` and the H8/300" in *Using ld*, for a fuller description.

`-mh` Generate code for the H8/300H.

4.14.17 Options for System V

These additional options are available on System V Release 4 for compatibility with other compilers on those systems:

`-Qy` Identify the versions of each tool used by the compiler, in a `.ident` assembler directive in the output.

`-Qn` Refrain from adding `.ident` directives to the output file (this is the default).

`-YP,dirs` Search the directories *dirs*, and no others, for libraries specified with '`-l`'.

`-Ym,dir` Look in the directory *dir* to find the M4 preprocessor. The assembler uses this option.

4.14.18 Zilog Z8000 Option

GNU CC recognizes one special option when configured to generate code for the Z8000 family:

`-mz8001` Generate code for the segmented variant of the Z8000 architecture. (Without this option, `gcc` generates unsegmented Z8000 code; suitable, for example, for the Z8002.)

4.14.19 Options for the H8/500

These options control some compilation choices specific to the Hitachi H8/500:

`-mspace` When a tradeoff is available between code size and speed, generate smaller code.

`-mspeed` When a tradeoff is available between code size and speed, generate faster code.

`-mint32` Make `int` data 32 bits by default.

`-mcode32` Compile code for a 32 bit address space.

`-mdata32` Compile data for a 32 bit address space.

`-mtiny` Compile both data and code sections using the same 16-bit address space.

`-msmall` Compile both data and code sections for 16-bit address spaces, but use different link segments.

`-mmedium` Compile code for a 32-bit address space, but data for a 16-bit address space. This is the same as specifying '`-mcode32`' *without* '`-mdata32`'.

`-mcompact`
Compile data for a 32-bit address space, but code for a 16-bit address space. This is the same as specifying '`-mdata32`' *without* '`-mcode32`'.

`-mbig` Compile both data and code sections for 32-bit address spaces. This is the same as specifying *both* '`-mdata32`' and '`-mcode32`'.

4.15 Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of '`-ffoo`' would be '`-fno-foo`'. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing '`no-`' or adding it.

`-fpcc-struct-return`
: Return "short" `struct` and `union` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing intercallability between GNU CC-compiled files and files compiled with other compilers.

  The precise convention for returning structures in memory depends on the target configuration macros.

  Short structures and unions are those whose size and alignment match that of some integer type.

`-freg-struct-return`
: Use the convention that `struct` and `union` values are returned in registers when possible. This is more efficient for small structures than '`-fpcc-struct-return`'.

  If you specify neither '`-fpcc-struct-return`' nor its contrary '`-freg-struct-return`', GNU CC defaults to whichever convention is standard for the target. If there is no standard convention, GNU CC defaults to '`-fpcc-struct-return`', except on targets where GNU CC is the principal compiler. In those cases, we can choose the standard, and we chose the more efficient register return alternative.

`-fshort-enums`
: Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type will be equivalent to the smallest integer type which has enough room.

`-fshort-double`
: Use the same size for `double` as for `float`.

`-fshared-data`
: Requests that the data and non-`const` variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

`-fno-common`
: Allocate even uninitialized global variables in the bss section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without `extern`) in two different compilations, you will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

`-fno-ident`
: Ignore the '`#ident`' directive.

`-fno-gnu-linker`
: Do not output global initializations (such as C++ constructors and destructors) in the form used by the GNU linker (on systems where the GNU linker is the standard method of handling them). Use this option when you want to use a non-GNU linker, which also requires using the `collect2` program to make sure the system linker includes constructors and destructors. (`collect2` is included in the GNU CC distribution.) For systems which *must* use `collect2`, the compiler driver `gcc` is configured to do this automatically.

`-finhibit-size-directive`
: Don't output a `.size` assembler directive, or anything else that would cause trouble if the function is split in the middle, and the two halves are placed at locations far apart in memory. This option is used when compiling '`crtstuff.c`'; you should not need to use it for anything else.

`-fverbose-asm`
: Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

`-fvolatile`
: Consider all memory references through pointers to be volatile.

`-fvolatile-global`
: Consider all memory references to extern and global data items to be volatile.

`-fpic`
: Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). If the GOT size for the linked

executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that '`-fpic`' does not work; in that case, recompile with '`-fPIC`' instead. (These maximums are 16k on the m88k, 8k on the Sparc, and 32k on the m68k and RS/6000. The 386 has no such limit.)

Position-independent code requires special support, and therefore works only on certain machines. For the 386, GNU CC supports PIC for System V but not for the Sun 386i. Code generated for the IBM RS/6000 is always position-independent.

The GNU assembler does not fully support PIC. Currently, you must use some other assembler in order for PIC to work. We would welcome volunteers to upgrade GAS to handle this; the first part of the job is to figure out what the assembler must do differently.

`-fPIC`
: If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k, m88k and the Sparc.

Position-independent code requires special support, and therefore works only on certain machines.

`-ffixed-`*reg*
: Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the `REGISTER_NAMES` macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-used-`*reg*
: Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-saved-reg`
: Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

  Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

  A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

  This flag does not have a negative form, because it specifies a three-way choice.

`-fpack-struct`
: Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code suboptimal, and the offsets of structure members won't agree with system libraries.

`+e0`
`+e1`
: Control whether virtual function definitions in classes are used to generate code, or only to define interfaces for their callers. (C++ only).

  These options are provided for compatibility with `cfront` 1.x usage; the recommended alternative GNU C++ usage is in flux. See Section 7.4 "Declarations and Definitions in One Header," page 191.

  With '`+e0`', virtual function definitions in classes are declared `extern`; the declaration is used only as an interface specification, not to generate code for the virtual functions (in this compilation).

  With '`+e1`', G++ actually generates the code implementing virtual functions defined in the code, and makes them publicly visible.

`-funaligned-pointers`
: Assume that all pointers contain unaligned addresses. On machines where unaligned memory accesses trap, this will result in much larger and slower code for all pointer dereferences, but the code will work even if addresses are unaligned.

`-funaligned-struct-hack`
: Always access structure fields using loads and stores of the declared size. This option is useful for code that derefences pointers to unaligned structures, but only accesses fields that

are themselves aligned. Without this option, gcc may try to use a memory access larger than the field. This might give an unaligned access fault on some hardware.

This option makes some invalid code work at the expense of disabling some optimizations. It is strongly recommended that this option not be used.

4.16 Environment Variables Affecting GNU CC

This section describes several environment variables that affect how GNU CC operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note that you can also specify places to search using options such as '-B', '-I' and '-L' (see Section 4.12 "Directory Options," page 64). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GNU CC.

`TMPDIR`
: If `TMPDIR` is set, it specifies the directory to use for temporary files. GNU CC uses temporary files to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

`GCC_EXEC_PREFIX`
: If `GCC_EXEC_PREFIX` is set, it specifies a prefix to use in the names of the subprograms executed by the compiler. No slash is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish.

  If GNU CC cannot find the subprogram using the specified prefix, it tries looking in the usual places for the subprogram.

  The default of `GCC_EXEC_PREFIX` is '*prefix*/lib/gcc-lib/', where *prefix* is the value of `prefix` when you ran the 'configure' script.

  Other prefixes specified with '-B' take precedence over this prefix.

  This prefix is also used for finding files such as 'crt0.o' that are used for linking.

  In addition, the prefix is used in an unusual way in finding the directories to search for header files. For each of the standard directories whose name normally begins with '/usr/local/lib/gcc-lib' (more precisely, with the value of

GCC_INCLUDE_DIR), GNU CC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with '-Bfoo/', GNU CC will search 'foo/bar' where it would normally search '/usr/local/lib/bar'. These alternate directories are searched first; the standard directories come next.

COMPILER_PATH
: The value of COMPILER_PATH is a colon-separated list of directories, much like PATH. GNU CC tries the directories thus specified when searching for subprograms, if it can't find the subprograms using GCC_EXEC_PREFIX.

LIBRARY_PATH
: The value of LIBRARY_PATH is a colon-separated list of directories, much like PATH. GNU CC tries the directories thus specified when searching for special linker files, if it can't find them using GCC_EXEC_PREFIX. Linking using GNU CC also uses these directories when searching for ordinary libraries for the '-l' option (but directories specified with '-L' come first).

C_INCLUDE_PATH
CPLUS_INCLUDE_PATH
OBJC_INCLUDE_PATH
: These environment variables pertain to particular languages. Each variable's value is a colon-separated list of directories, much like PATH. When GNU CC searches for header files, it tries the directories listed in the variable for the language you are using, after the directories specified with '-I' but before the standard header file directories.

DEPENDENCIES_OUTPUT
: If this variable is set, its value specifies how to output dependencies for Make based on the header files processed by the compiler. This output looks much like the output from the '-M' option (see Section 4.9 "Preprocessor Options," page 58), but it goes to a separate file, and is in addition to the usual results of compilation.

  The value of DEPENDENCIES_OUTPUT can be just a file name, in which case the Make rules are written to that file, guessing the target name from the source file name. Or the value can have the form '*file target*', in which case the rules are written to file *file* using *target* as the target name.

4.17 Running Protoize

The program `protoize` is an optional part of GNU C. You can use it to add prototypes to a program, thus converting the program to ANSI C in one respect. The companion program `unprotoize` does the reverse: it removes argument types from any prototypes that are found.

When you run these programs, you must specify a set of source files as command line arguments. The conversion programs start out by compiling these files to see what functions they define. The information gathered about a file *foo* is saved in a file named '*foo*.X'.

After scanning comes actual conversion. The specified files are all eligible to be converted; any files they include (whether sources or just headers) are eligible as well.

But not all the eligible files are converted. By default, `protoize` and `unprotoize` convert only source and header files in the current directory. You can specify additional directories whose files should be converted with the '-d *directory*' option. You can also specify particular files to exclude with the '-x *file*' option. A file is converted if it is eligible, its directory name matches one of the specified directory names, and its name within the directory has not been excluded.

Basic conversion with `protoize` consists of rewriting most function definitions and function declarations to specify the types of the arguments. The only ones not rewritten are those for varargs functions.

`protoize` optionally inserts prototype declarations at the beginning of the source file, to make them available for any calls that precede the function's definition. Or it can insert prototype declarations with block scope in the blocks where undeclared functions are called.

Basic conversion with `unprotoize` consists of rewriting most function declarations to remove any argument types, and rewriting function definitions to the old-style pre-ANSI form.

Both conversion programs print a warning for any function declaration or definition that they can't convert. You can suppress these warnings with '-q'.

The output from `protoize` or `unprotoize` replaces the original source file. The original file is renamed to a name ending with '.save'. If the '.save' file already exists, then the source file is simply discarded.

`protoize` and `unprotoize` both depend on GNU CC itself to scan the program and collect information about the functions it uses. So neither of these programs will work until GNU CC is installed.

Here is a table of the options you can use with `protoize` and `unprotoize`. Each option works with both programs unless otherwise stated.

`-B directory`
: Look for the file '`SYSCALLS.c.X`' in *directory*, instead of the usual directory (normally '`/usr/local/lib`'). This file contains prototype information about standard system functions. This option applies only to `protoize`.

`-c compilation-options`
: Use *compilation-options* as the options when running `gcc` to produce the '`.X`' files. The special option '`-aux-info`' is always passed in addition, to tell `gcc` to write a '`.X`' file.

Note that the compilation options must be given as a single argument to `protoize` or `unprotoize`. If you want to specify several `gcc` options, you must quote the entire set of compilation options to make them a single word in the shell.

There are certain `gcc` arguments that you cannot use, because they would produce the wrong kind of output. These include '`-g`', '`-O`', '`-c`', '`-S`', and '`-o`' If you include these in the *compilation-options*, they are ignored.

`-C`
: Rename files to end in '`.C`' instead of '`.c`'. This is convenient if you are converting a C program to C++. This option applies only to `protoize`.

`-g`
: Add explicit global declarations. This means inserting explicit declarations at the beginning of each source file for each function that is called in the file and was not declared. These declarations precede the first function definition that contains a call to an undeclared function. This option applies only to `protoize`.

`-i string`
: Indent old-style parameter declarations with the string *string*. This option applies only to `protoize`.

`unprotoize` converts prototyped function definitions to old-style function definitions, where the arguments are declared between the argument list and the initial '`{`'. By default, `unprotoize` uses five spaces as the indentation. If you want to indent with just one space instead, use '`-i " "`'.

`-k`
: Keep the '`.X`' files. Normally, they are deleted after conversion is finished.

`-l`
: Add explicit local declarations. `protoize` with '`-l`' inserts a prototype declaration for each function in each block which calls the function without any declaration. This option applies only to `protoize`.

`-n` Make no real changes. This mode just prints information about the conversions that would have been done without '`-n`'.

`-N` Make no '`.save`' files. The original files are simply deleted. Use this option with caution.

`-p` *program*
Use the program *program* as the compiler. Normally, the name '`gcc`' is used.

`-q` Work quietly. Most warnings are suppressed.

`-v` Print the version number, just like '`-v`' for `gcc`.

If you need special compiler options to compile one of your program's source files, then you should generate that file's '`.X`' file specially, by running `gcc` on that source file with the appropriate options and the option '`-aux-info`'. Then run `protoize` on the entire set of files. `protoize` will use the existing '`.X`' file because it is newer than the source file. For example:

```
gcc -Dfoo=bar file1.c -aux-info
protoize *.c
```

You need to include the special files along with the rest in the `protoize` command, even though their '`.X`' files already exist, because otherwise they won't get converted.

See Section 9.10 "Protoize Caveats," page 225, for more information on how to use `protoize` successfully.

5 Installing GNU CC

Here is the procedure for installing GNU CC on a Unix system. See Section 5.5 “VMS Install,” page 133, for VMS systems. In this section we assume you compile in the same directory that contains the source files; see Section 5.2 “Other Dir,” page 127, to find out how to compile in a separate directory on Unix systems.

You cannot install GNU C by itself on MSDOS; it will not compile under any MSDOS compiler except itself. You need to get the complete compilation package DJGPP, which includes binaries as well as sources, and includes all the necessary compilation tools and libraries.

1. If you have built GNU CC previously in the same directory for a different target machine, do '`make distclean`' to delete all files that might be invalid. One of the files this deletes is '`Makefile`'; if '`make distclean`' complains that '`Makefile`' does not exist, it probably means that the directory is already suitably clean.
2. On a System V release 4 system, make sure '`/usr/bin`' precedes '`/usr/ucb`' in `PATH`. The `cc` command in '`/usr/ucb`' uses libraries which have bugs.
3. Specify the host, build and target machine configurations. You do this by running the file '`configure`'.

   The *build* machine is the system which you are using, the *host* machine is the system where you want to run the resulting compiler (normally the build machine), and the *target* machine is the system for which you want the compiler to generate code.

   If you are building a compiler to produce code for the machine it runs on (a native compiler), you normally do not need to specify any operands to '`configure`'; it will try to guess the type of machine you are on and use that as the build, host and target machines. So you don't need to specify a configuration when building a native compiler unless '`configure`' cannot figure out what your configuration is or guesses wrong.

   In those cases, specify the build machine's *configuration name* with the '`--build`' option; the host and target will default to be the same as the build machine. (If you are building a cross-compiler, see Section 5.3 “Cross-Compiler,” page 127.)

   Here is an example:

   from an MSDOS console window or from the program manager dialog box. Configure.bat assumes that you have already installed and in your path a Unix-like sed program which is used to modify Makefile.in to create a working Makefile.

   ```
   ./configure --build=sparc-sun-sunos4.1
   ```

A configuration name may be canonical or it may be more or less abbreviated.

A canonical configuration name has three parts, separated by dashes. It looks like this: '*cpu-company-system*'. (The three parts may themselves contain dashes; 'configure' can figure out which dashes serve which purpose.) For example, 'm68k-sun-sunos4.1' specifies a Sun 3.

You can also replace parts of the configuration by nicknames or aliases. For example, 'sun3' stands for 'm68k-sun', so 'sun3-sunos4.1' is another way to specify a Sun 3. You can also use simply 'sun3-sunos', since the version of SunOS is assumed by default to be version 4. 'sun3-bsd' also works, since 'configure' knows that the only BSD variant on a Sun 3 is SunOS.

You can specify a version number after any of the system types, and some of the CPU types. In most cases, the version is irrelevant, and will be ignored. So you might as well specify the version if you know it.

See Section 5.1 "Configurations," page 111, for a list of supported configuration names and notes on many of the configurations. You should check the notes in that section before proceding any further with the installation of GNU CC.

There are four additional options you can specify independently to describe variant hardware and software configurations. These are '--with-gnu-as', '--with-gnu-ld', '--with-stabs' and '--nfp'.

'--with-gnu-as'

If you will use GNU CC with the GNU assembler (GAS), you should declare this by using the '--with-gnu-as' option when you run 'configure'.

Using this option does not install GAS. It only modifies the output of GNU CC to work with GAS. Building and installing GAS is up to you.

Conversely, if you *do not* wish to use GAS and do not specify '--with-gnu-as' when building GNU CC, it is up to you to make sure that GAS is not installed. GNU CC searches for a program named as in various directories; if the program it finds is GAS, then it runs GAS. If you are not sure where GNU CC finds the assembler it is using, try specifying '-v' when you run it.

The systems where it makes a difference whether you use GAS are

'hppa1.0-*any*-*any*', 'hppa1.1-*any*-*any*', 'i386-*any*-sysv', 'i386-*any*-isc',

'i860-*any*-bsd', 'm68k-bull-sysv', 'm68k-hp-hpux', 'm68k-sony-bsd',
'm68k-altos-sysv', 'm68000-hp-hpux', 'm68000-att-sysv', '*any*-lynx-lynxos', and 'mips-*any*'). On any other system, '--with-gnu-as' has no effect.

On the systems listed above (except for the HP-PA, for ISC on the 386, and for 'mips-sgi-irix5.*'), if you use GAS, you should also use the GNU linker (and specify '--with-gnu-ld').

'--with-gnu-ld'
: Specify the option '--with-gnu-ld' if you plan to use the GNU linker with GNU CC.

This option does not cause the GNU linker to be installed; it just modifies the behavior of GNU CC to work with the GNU linker. Specifically, it inhibits the installation of `collect2`, a program which otherwise serves as a front-end for the system's linker on most configurations.

'--with-stabs'
: On MIPS based systems and on Alphas, you must specify whether you want GNU CC to create the normal ECOFF debugging format, or to use BSD-style stabs passed through the ECOFF symbol table. The normal ECOFF debug format cannot fully handle languages other than C. BSD stabs format can handle other languages, but it only works with the GNU debugger GDB.

Normally, GNU CC uses the ECOFF debugging format by default; if you prefer BSD stabs, specify '--with-stabs' when you configure GNU CC.

No matter which default you choose when you configure GNU CC, the user can use the '-gcoff' and '-gstabs+' options to specify explicitly the debug format for a particular compilation.

'--with-stabs' is meaningful on the ISC system on the 386, also, if '--with-gas' is used. It selects use of stabs debugging information embedded in COFF output. This kind of debugging information supports C++ well; ordinary COFF debugging information does not.

'--with-stabs' is also meaningful on 386 systems running SVR4. It selects use of stabs debugging information embedded in ELF output. The C++ compiler currently (2.6.0) does not support the DWARF debugging information normally used on 386 SVR4 platforms; stabs provide

a workable alternative. This requires gas and gdb, as the normal SVR4 tools can not generate or interpret stabs.

'--nfp' On certain systems, you must specify whether the machine has a floating point unit. These systems include 'm68k-sun-sunos*n*' and 'm68k-isi-bsd'. On any other system, '--nfp' currently has no effect, though perhaps there are other systems where it could usefully make a difference.

The 'configure' script searches subdirectories of the source directory for other compilers that are to be integrated into GNU CC. The GNU compiler for C++, called G++ is in a subdirectory named 'cp'. 'configure' inserts rules into 'Makefile' to build all of those compilers.

Here we spell out what files will be set up by configure. Normally you need not be concerned with these files.

- A symbolic link named 'config.h' is made to the top-level config file for the machine you plan to run the compiler on (see section "The Configuration File" in *Using and Porting GCC*). This file is responsible for defining information about the host machine. It includes 'tm.h'.

  The top-level config file is located in the subdirectory 'config'. Its name is always 'xm-*something*.h'; usually 'xm-*machine*.h', but there are some exceptions.

  If your system does not support symbolic links, you might want to set up 'config.h' to contain a '#include' command which refers to the appropriate file.
- A symbolic link named 'tconfig.h' is made to the top-level config file for your target machine. This is used for compiling certain programs to run on that machine.
- A symbolic link named 'tm.h' is made to the machine-description macro file for your target machine. It should be in the subdirectory 'config' and its name is often '*machine*.h'.
- A symbolic link named 'md' will be made to the machine description pattern file. It should be in the 'config' subdirectory and its name should be '*machine*.md'; but *machine* is often not the same as the name used in the 'tm.h' file because the 'md' files are more general.
- A symbolic link named 'aux-output.c' will be made to the output subroutine file for your machine. It should be in the 'config' subdirectory and its name should be '*machine*.c'.
- The command file 'configure' also constructs the file 'Makefile' by adding some text to the template file 'Makefile.in'. The

additional text comes from files in the 'config' directory, named 't-*target*' and 'x-*host*'. If these files do not exist, it means nothing needs to be added for a given target or host.

4. The standard directory for installing GNU CC is '/usr/local/lib'. If you want to install its files somewhere else, specify '--prefix=*dir*' when you run 'configure'. Here *dir* is a directory name to use instead of '/usr/local' for all purposes with one exception: the directory '/usr/local/include' is searched for header files no matter where you install the compiler. To override this name, use the --local-prefix option below.
5. Specify '--local-prefix=*dir*' if you want the compiler to search directory '*dir*/include' for locally installed header files *instead* of '/usr/local/include'.

   You should specify '--local-prefix' **only** if your site has a different convention (not '/usr/local') for where to put site-specific files.

   Do not specify '/usr' as the '--local-prefix'! The directory you use for '--local-prefix' **must not** contain any of the system's standard header files. If it did contain them, certain programs would be miscompiled (including GNU Emacs, on certain targets), because this would override and nullify the header file corrections made by the fixincludes script.
6. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output files 'c-parse.c' and 'cexp.c' are more recent than 'c-parse.y' and 'cexp.y' and you do not plan to change the '.y' files.)

   Bison versions older than Sept 8, 1988 will produce incorrect output for 'c-parse.c'.
7. If you have chosen a configuration for GNU CC which requires other GNU tools (such as GAS or the GNU linker) instead of the standard system tools, install the required tools in the build directory under the names 'as', 'ld' or whatever is appropriate. This will enable the compiler to find the proper tools for compilation of the program 'enquire'.

   Alternatively, you can do subsequent compilation using a value of the PATH environment variable such that the necessary GNU tools come before the standard system tools.
8. Build the compiler. Just type 'make LANGUAGES=c' in the compiler directory.

   'LANGUAGES=c' specifies that only the C compiler should be compiled. The makefile normally builds compilers for all the supported languages; currently, C, C++ and Objective C. However, C is the only language that is sure to work when you build with other non-GNU

C compilers. In addition, building anything but C at this stage is a waste of time.

In general, you can specify the languages to build by typing the argument 'LANGUAGES="*list*"', where *list* is one or more words from the list 'c', 'c++', and 'objective-c'. If you have any additional GNU compilers as subdirectories of the GNU CC source directory, you may also specify their names in this list.

Ignore any warnings you may see about "statement not reached" in 'insn-emit.c'; they are normal. Also, warnings about "unknown escape sequence" are normal in 'genopinit.c' and perhaps some other files. Likewise, you should ignore warnings about "constant is so large that it is unsigned" in 'insn-emit.c' and 'insn-recog.c'. Any other compilation errors may represent bugs in the port to your machine or operating system, and should be investigated and reported (see Chapter 10 "Bugs," page 233).

Some commercial compilers fail to compile GNU CC because they have bugs or limitations. For example, the Microsoft compiler is said to run out of macro space. Some Ultrix compilers run out of expression space; then you need to break up the statement where the problem happens.

9. If you are building a cross-compiler, stop here. See Section 5.3 "Cross-Compiler," page 127.
10. Move the first-stage object files and executables into a subdirectory with this command:

```
make stage1
```

    The files are moved into a subdirectory named 'stage1'. Once installation is complete, you may wish to delete these files with `rm -r stage1`.
11. If you have chosen a configuration for GNU CC which requires other GNU tools (such as GAS or the GNU linker) instead of the standard system tools, install the required tools in the 'stage1' subdirectory under the names 'as', 'ld' or whatever is appropriate. This will enable the stage 1 compiler to find the proper tools in the following stage.

    Alternatively, you can do subsequent compilation using a value of the PATH environment variable such that the necessary GNU tools come before the standard system tools.
12. Recompile the compiler with itself, with this command:

```
make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O"
```

    This is called making the stage 2 compiler.

    The command shown above builds compilers for all the supported languages. If you don't want them all, you can specify the languages

to build by typing the argument 'LANGUAGES="*list*"'. *list* should contain one or more words from the list 'c', 'c++', 'objective-c', and 'proto'. Separate the words with spaces. 'proto' stands for the programs protoize and unprotoize; they are not a separate language, but you use LANGUAGES to enable or disable their installation.

If you are going to build the stage 3 compiler, then you might want to build only the C language in stage 2.

Once you have built the stage 2 compiler, if you are short of disk space, you can delete the subdirectory 'stage1'.

On a 68000 or 68020 system lacking floating point hardware, unless you have selected a 'tm.h' file that expects by default that there is no such hardware, do this instead:

```
make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O -msoft-float"
```

13. If you wish to test the compiler by compiling it with itself one more time, install any other necessary GNU tools (such as GAS or the GNU linker) in the 'stage2' subdirectory as you did in the 'stage1' subdirectory, then do this:

    ```
    make stage2
    make CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O"
    ```

    This is called making the stage 3 compiler. Aside from the '-B' option, the compiler options should be the same as when you made the stage 2 compiler. But the LANGUAGES option need not be the same. The command shown above builds compilers for all the supported languages; if you don't want them all, you can specify the languages to build by typing the argument 'LANGUAGES="*list*"', as described above.

    If you do not have to install any additional GNU tools, you may use the command

    ```
    make bootstrap LANGUAGES=language-list
            BOOT_CFLAGS=option-list
    ```

    instead of making 'stage1', 'stage2', and performing the two compiler builds.

14. Then compare the latest object files with the stage 2 object files—they ought to be identical, aside from time stamps (if any).

    On some systems, meaningful comparison of object files is impossible; they always appear "different." This is currently true on Solaris and probably on all systems that use ELF object file format. On some versions of Irix on SGI machines and OSF/1 on Alpha systems, you will not be able to compare the files without specifying '-save-temps'; see the description of individual systems above to see if you get comparison failures. You may have similar problems on other systems.

    Use this command to compare the files:

```
make compare
```

This will mention any object files that differ between stage 2 and stage 3. Any difference, no matter how innocuous, indicates that the stage 2 compiler has compiled GNU CC incorrectly, and is therefore a potentially serious bug which you should investigate and report (see Chapter 10 “Bugs,” page 233).

If your system does not put time stamps in the object files, then this is a faster way to compare them (using the Bourne shell):

```
for file in *.o; do
cmp $file stage2/$file
done
```

If you have built the compiler with the ‘`-mno-mips-tfile`’ option on MIPS machines, you will not be able to compare the files.

15. Build the Objective C library (if you have built the Objective C compiler). Here is the command to do this:

```
make objc-runtime CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O"
```

16. Install the compiler driver, the compiler's passes and run-time support with ‘`make install`’. Use the same value for `CC`, `CFLAGS` and `LANGUAGES` that you used when compiling the files that are being installed. One reason this is necessary is that some versions of Make have bugs and recompile files gratuitously when you do this step. If you use the same variable values, those files will be recompiled properly.

For example, if you have built the stage 2 compiler, you can use the following command:

```
make install CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O"
            LANGUAGES="list"
```

This command copies the three files ‘`cc1`’, ‘`cpp`’ and ‘`libgcc.a`’ into the files ‘`cc1`’, ‘`cpp`’ and ‘`libgcc.a`’ and puts them in the directory ‘`/usr/local/lib/gcc-lib/`*`target`*`/`*`version`*’, which is where the compiler driver program looks for them. Here *`target`* is the target machine type specified when you ran ‘`configure`’, and *`version`* is the version number of GNU CC. This naming scheme permits various versions and/or cross-compilers to coexist.

This also copies the driver program ‘`xgcc`’ into ‘`/usr/local/bin/gcc`’, so that it appears in typical execution search paths.

On some systems, this command causes recompilation of some files. This is usually due to bugs in `make`. You should either ignore this problem, or use GNU Make.

Warning: there is a bug in `alloca` in the Sun library. To avoid this bug, be sure to install the executables of GNU CC that were compiled by GNU CC. (That is, the executables

from stage 2 or 3, not stage 1.) They use `alloca` as a built-in function and never the one in the library.

(It is usually better to install GNU CC executables from stage 2 or 3, since they usually run faster than the ones compiled with some other compiler.)

17. Install the Objective C library (if you are installing the Objective C compiler). Here is the command to do this:

```
make install-libobjc CC="stage2/xgcc -Bstage2/" CFLAGS="-g -
O"
```

18. If you're going to use C++, it's likely that you need to also install the libg++ distribution. It should be available from the same place where you got the GNU C distribution. Just as GNU C does not distribute a C runtime library, it also does not include a C++ runtime library. All I/O functionality, special class libraries, etc., are available in the libg++ distribution.

5.1 Configurations Supported by GNU CC

Here are the possible CPU types:

> 1750a, a29k, alpha, arm, c*n*, clipper, dsp16xx, elxsi, h8300, hppa1.0, hppa1.1, i370, i386, i486, i586, i860, i960, m68000, m68k, m88k, mips, mipsel, mips64, mips64el, ns32k, powerpc, pyramid, romp, rs6000, sh, sparc, sparclite, sparc64, vax, we32k.

Here are the recognized company names. As you can see, customary abbreviations are used rather than the longer official names.

> acorn, alliant, altos, apollo, att, bull, cbm, convergent, convex, crds, dec, dg, dolphin, elxsi, encore, harris, hitachi, hp, ibm, intergraph, isi, mips, motorola, ncr, next, ns, omron, plexus, sequent, sgi, sony, sun, tti, unicom, wrs.

The company name is meaningful only to disambiguate when the rest of the information supplied is insufficient. You can omit it, writing just '*cpu-system*', if it is not needed. For example, '`vax-ultrix4.2`' is equivalent to '`vax-dec-ultrix4.2`'.

Here is a list of system types:

> 386bsd, aix, acis, amigados, aos, aout, bosx, bsd, clix, coff, ctix, cxux, dgux, dynix, ebmon, ecoff, elf, esix, freebsd, hms, genix, gnu, gnu/linux, hiux, hpux, iris, irix, isc, luna, lynxos, mach, minix, msdos, mvs, netbsd, newsos, nindy, ns, osf, osfrose, ptx, riscix, riscos, rtu, sco, sim, solaris, sunos, sym, sysv, udi, ultrix, unicos, uniplus, unos, vms, vsta, vxworks, winnt, xenix.

You can omit the system type; then 'configure' guesses the operating system from the CPU and company.

You can add a version number to the system type; this may or may not make a difference. For example, you can write 'bsd4.3' or 'bsd4.4' to distinguish versions of BSD. In practice, the version number is most needed for 'sysv3' and 'sysv4', which are often treated differently.

If you specify an impossible combination such as 'i860-dg-vms', then you may get an error message from 'configure', or it may ignore part of the information and do the best it can with the rest. 'configure' always prints the canonical name for the alternative that it used. GNU CC does not support all possible alternatives.

Often a particular model of machine has a name. Many machine names are recognized as aliases for CPU/company combinations. Thus, the machine name 'sun3', mentioned above, is an alias for 'm68k-sun'. Sometimes we accept a company name as a machine name, when the name is popularly used for a particular machine. Here is a table of the known machine names:

> 3300, 3b1, 3b*n*, 7300, altos3068, altos, apollo68, att-7300, balance, convex-c*n*, crds, decstation-3100, decstation, delta, encore, fx2800, gmicro, hp7*nn*, hp8*nn*, hp9k2*nn*, hp9k3*nn*, hp9k7*nn*, hp9k8*nn*, iris4d, iris, isi68, m3230, magnum, merlin, miniframe, mmax, news-3600, news800, news, next, pbd, pc532, pmax, powerpc, ps2, risc-news, rtpc, sun2, sun386i, sun386, sun3, sun4, symmetry, tower-32, tower.

Remember that a machine name specifies both the cpu type and the company name. If you want to install your own homemade configuration files, you can use 'local' as the company name to access them. If you use configuration '*cpu*-local', the configuration name without the cpu prefix is used to form the configuration file names.

Thus, if you specify 'm68k-local', configuration uses files 'm68k.md', 'local.h', 'm68k.c', 'xm-local.h', 't-local', and 'x-local', all in the directory 'config/m68k'.

Here is a list of configurations that have special treatment or special things you must know:

'1750a-*-*'

MIL-STD-1750A processors.

Starting with GCC 2.6.1, the MIL-STD-1750A cross configuration no longer supports the Tektronix Assembler, but instead produces output for as1750, an assembler/linker available under the GNU Public License for the 1750A. Contact *okellogg@salyko.cube.net* for more details on obtaining

'as1750'. A similarly licensed simulator for the 1750A is available from same address.

You should ignore a fatal error during the building of libgcc (libgcc is not yet implemented for the 1750A.)

The as1750 assembler requires the file 'ms1750.inc', which is found in the directory 'config/1750a'.

GNU CC produced the same sections as the Fairchild F9450 C Compiler, namely:

NREL
: The program code section.

SREL
: The read/write (RAM) data section.

KREL
: The read-only (ROM) constants section.

IREL
: Initialization section (code to copy KREL to SREL).

The smallest addressable unit is 16 bits (BITS_PER_UNIT is 16). This means that type 'char' is represented with a 16-bit word per character. The 1750A's "Load/Store Upper/Lower Byte" instructions are not used by GNU CC.

There is a problem with long argument lists to functions. The compiler aborts if the sum of space needed by all arguments exceeds 14 words. This is because the arguments are passed in registers (R0..R13) not on the stack, and there is a problem with passing further arguments (i.e. beyond those in R0..R13) via the stack.

If efficiency is less important than using long argument lists, you can change the definition of the FUNCTION_ARG macro in 'config/1750/1750a.h' to always return zero. If you do that, GNU CC will pass all parameters on the stack.

'alpha-*-osf1'
: Systems using processors that implement the DEC Alpha architecture and are running the OSF/1 operating system, for example the DEC Alpha AXP systems. (VMS on the Alpha is not currently supported by GNU CC.)

GNU CC writes a '.verstamp' directive to the assembler output file unless it is built as a cross-compiler. It gets the version to use from the system header file '/usr/include/stamp.h'. If you install a new version of OSF/1, you should rebuild GCC to pick up the new version stamp.

Note that since the Alpha is a 64-bit architecture, cross-compilers from 32-bit machines will not generate code as

efficient as that generated when the compiler is running on a 64-bit machine because many optimizations that depend on being able to represent a word on the target in an integral value on the host cannot be performed. Building cross-compilers on the Alpha for 32-bit machines has only been tested in a few cases and may not work properly.

`make compare` may fail on old versions of OSF/1 unless you add '`-save-temps`' to `CFLAGS`. On these systems, the name of the assembler input file is stored in the object file, and that makes comparison fail if it differs between the `stage1` and `stage2` compilations. The option '`-save-temps`' forces a fixed name to be used for the assembler input file, instead of a randomly chosen name in '`/tmp`'. Do not add '`-save-temps`' unless the comparisons fail without that option. If you add '`-save-temps`', you will have to manually delete the '`.i`' and '`.s`' files after each series of compilations.

GNU CC now supports both the native (ECOFF) debugging format used by DBX and GDB and an encapsulated STABS format for use only with GDB. See the discussion of the '`--with-stabs`' option of '`configure`' above for more information on these formats and how to select them.

There is a bug in DEC's assembler that produces incorrect line numbers for ECOFF format when the '`.align`' directive is used. To work around this problem, GNU CC will not emit such alignment directives while writing ECOFF format debugging information even if optimization is being performed. Unfortunately, this has the very undesirable side-effect that code addresses when '`-O`' is specified are different depending on whether or not '`-g`' is also specified.

To avoid this behavior, specify '`-gstabs+`' and use GDB instead of DBX. DEC is now aware of this problem with the assembler and hopes to provide a fix shortly.

'`arm`'
: Advanced RISC Machines ARM-family processors. These are often used in embedded applications. There are no standard Unix configurations. This configuration corresponds to the basic instruction sequences and will produce a.out format object modules.

  You may need to make a variant of the file '`arm.h`' for your particular configuration.

'`arm-*-riscix`'
: The ARM2 or ARM3 processor running RISC iX, Acorn's port of BSD Unix. If you are running a version of RISC iX prior to

1.2 then you must specify the version number during configuration. Note that the assembler shipped with RISC iX does not support stabs debugging information; a new version of the assembler, with stabs support included, is now available from Acorn.

'a29k'
: AMD Am29k-family processors. These are normally used in embedded applications. There are no standard Unix configurations. This configuration corresponds to AMD's standard calling sequence and binary interface and is compatible with other 29k tools.

  You may need to make a variant of the file 'a29k.h' for your particular configuration.

'a29k-*-bsd'
: AMD Am29050 used in a system running a variant of BSD Unix.

'decstation-*'
: DECstations can support three different personalities: Ultrix, DEC OSF/1, and OSF/rose. To configure GCC for these platforms use the following configurations:

  'decstation-ultrix'
  : Ultrix configuration.

  'decstation-osf1'
  : Dec's version of OSF/1.

  'decstation-osfrose'
  : Open Software Foundation reference port of OSF/1 which uses the OSF/rose object file format instead of ECOFF. Normally, you would not select this configuration.

  The MIPS C compiler needs to be told to increase its table size for switch statements with the '-Wf,-XNg1500' option in order to compile 'cp/parse.c'. If you use the '-O2' optimization option, you also need to use '-Olimit 3000'. Both of these options are automatically generated in the 'Makefile' that the shell script 'configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add '-Wf,-XNg1500 -Olimit 3000'.

'elxsi-elxsi-bsd'
: The Elxsi's C compiler has known limitations that prevent it from compiling GNU C. Please contact mrs@cygnus.com for more details.

'dsp16xx' A port to the AT&T DSP1610 family of processors.

'h8300-*-*'
The calling convention and structure layout has changed in release 2.6. All code must be recompiled. The calling convention now passes the first three arguments in function calls in registers. Structures are no longer a multiple of 2 bytes.

'hppa*-*-*'
There are two variants of this CPU, called 1.0 and 1.1, which have different machine descriptions. You must use the right one for your machine. All 7*nn* machines and 8*n*7 machines use 1.1, while all other 8*nn* machines use 1.0.

The easiest way to handle this problem is to use 'configure hp*nnn*' or 'configure hp*nnn*-hpux', where *nnn* is the model number of the machine. Then 'configure' will figure out if the machine is a 1.0 or 1.1. Use 'uname -a' to find out the model number of your machine.

'-g' does not work on HP-UX, since that system uses a peculiar debugging format which GNU CC does not know about. However, '-g' will work if you also use GAS and GDB in conjunction with GCC. We highly recommend using GAS for all HP-PA configurations.

You should be using GAS-2.3 (or later) along with GDB-4.12 (or later). These can be retrieved from all the traditional GNU ftp archive sites.

Build GAS and install the resulting binary as:

```
/usr/local/lib/gcc-lib/configuration/gccversion/as
```

where *configuration* is the configuration name (perhaps 'hp*nnn*-hpux') and *gccversion* is the GNU CC version number. Do this *before* starting the build process, otherwise you will get errors from the HPUX assembler while building 'libgcc2.a'. The command

```
make install-dir
```

will create the necessary directory hierarchy so you can install GAS before building GCC.

To enable debugging, configure GNU CC with the '--with-gnu-as' option before building.

It has been reported that GNU CC produces invalid assembly code for 1.1 machines running HP-UX 8.02 when using the HP assembler. Typically the errors look like this:

```
as: bug.s @line#15 [err#1060]
  Argument 0 or 2 in FARG upper
         - lookahead = ARGW1=FR,RTNVAL=GR
```

```
as: foo.s @line#28 [err#1060]
  Argument 0 or 2 in FARG upper
         - lookahead = ARGW1=FR
```

You can check the version of HP-UX you are running by executing the command 'uname -r'. If you are indeed running HP-UX 8.02 on a PA and using the HP assembler then configure GCC with "hp*nnn*-hpux8.02".

'i370-*-*'
: This port is very preliminary and has many known bugs. We hope to have a higher-quality port for this machine soon.

'i386-*-linuxaout'
: Use this configuration to generate a.out binaries on Linux. This is an obsolete configuration.

'i386-*-linux'
: Use this configuration to generate ELF binaries on Linux. You must use gas/binutils version 2.5.2 or later.

'i386-*-sco'
: Compilation with RCC is recommended. Also, it may be a good idea to link with GNU malloc instead of the malloc that comes with the system.

'i386-*-sco3.2v4'
: Use this configuration for SCO release 3.2 version 4.

'i386-*-isc'
: It may be a good idea to link with GNU malloc instead of the malloc that comes with the system.

In ISC version 4.1, 'sed' core dumps when building 'deduced.h'. Use the version of 'sed' from version 4.0.

'i386-*-esix'
: It may be good idea to link with GNU malloc instead of the malloc that comes with the system.

'i386-ibm-aix'
: You need to use GAS version 2.1 or later, and and LD from GNU binutils version 2.2 or later.

'i386-sequent-bsd'
: Go to the Berkeley universe before compiling. In addition, you probably need to create a file named 'string.h' containing just one line: '#include <strings.h>'.

'i386-sequent-ptx1*'
: Sequent DYNIX/ptx 1.x.

'i386-sequent-ptx2*'
: Sequent DYNIX/ptx 2.x.

'i386-sun-sunos4'
: You may find that you need another version of GNU CC to begin bootstrapping with, since the current version when built with the system's own compiler seems to get an infinite loop compiling part of 'libgcc2.c'. GNU CC version 2 compiled with GNU CC (any version) seems not to have this problem.

  See Section 5.4 "Sun Install," page 133, for information on installing GNU CC on Sun systems.

'i[345]86-*-winnt3.5'
: This version requires a GAS that has not let been released. Until it is, you can get a prebuilt binary version via anonymous ftp from 'cs.washington.edu:pub/gnat' or 'cs.nyu.edu:pub/gnat'. You must also use the Microsoft header files from the Windows NT 3.5 SDK. Find these on the CDROM in the '/mstools/h' directory dated 9/4/94. You must use a fixed version of Microsoft linker made especially for NT 3.5, which is also is available on the NT 3.5 SDK CDROM. If you do not have this linker, can you also use the linker from Visual C/C++ 1.0 or 2.0.

  Installing GNU CC for NT builds a wrapper linker, called 'ld.exe', which mimics the behaviour of Unix 'ld' in the specification of libraries ('-L' and '-l'). 'ld.exe' looks for both Unix and Microsoft named libraries. For example, if you specify '-lfoo', 'ld.exe' will look first for 'libfoo.a' and then for 'foo.lib'.

  You may install GNU CC for Windows NT in one of two ways, depending on whether or not you have a Unix-like shell and various Unix-like utilities.

  1. If you do not have a Unix-like shell and few Unix-like utilities, you will use a DOS style batch script called 'configure.bat'. Invoke it as configure winnt from an MSDOS console window or from the program manager dialog box. 'configure.bat' assumes you have already installed and have in your path a Unix-like 'sed' program which is used to create a working 'Makefile' from 'Makefile.in'.

     'Makefile' uses the Microsoft Nmake program maintenance utility and the Visual C/C++ V8.00 compiler to build GNU CC. You need only have the utilities 'sed' and 'touch' to use this installation method, which only automatically builds the compiler itself. You must then

examine what 'fixinc.winnt' does, edit the header files by hand and build 'libgcc.a' manually.

2. The second type of installation assumes you are running a Unix-like shell, have a complete suite of Unix-like utilities in your path, and have a previous version of GNU CC already installed, either through building it via the above installation method or acquiring a pre-built binary. In this case, use the 'configure' script in the normal fashion.

'i860-intel-osf1'
: This is the Paragon. If you have version 1.0 of the operating system, see Section 9.2 "Installation Problems," page 205, for special things you need to do to compensate for peculiarities in the system.

'*-lynx-lynxos'
: LynxOS 2.2 and earlier comes with GNU CC 1.x already installed as '/bin/gcc'. You should compile with this instead of '/bin/cc'. You can tell GNU CC to use the GNU assembler and linker, by specifying '--with-gnu-as --with-gnu-ld' when configuring. These will produce COFF format object files and executables; otherwise GNU CC will use the installed tools, which produce a.out format executables.

'm68000-hp-bsd'
: HP 9000 series 200 running BSD. Note that the C compiler that comes with this system cannot compile GNU CC; contact law@cs.utah.edu to get binaries of GNU CC for bootstrapping.

'm68k-altos'
: Altos 3068. You must use the GNU assembler, linker and debugger. Also, you must fix a kernel bug. Details in the file 'README.ALTOS'.

'm68k-att-sysv'
: AT&T 3b1, a.k.a. 7300 PC. Special procedures are needed to compile GNU CC with this machine's standard C compiler, due to bugs in that compiler. You can bootstrap it more easily with previous versions of GNU CC if you have them.

Installing GNU CC on the 3b1 is difficult if you do not already have GNU CC running, due to bugs in the installed C compiler. However, the following procedure might work. We are unable to test it.

1. Comment out the '#include "config.h"' line on line 37 of 'cccp.c' and do 'make cpp'. This makes a preliminary version of GNU cpp.
2. Save the old '/lib/cpp' and copy the preliminary GNU cpp to that file name.
3. Undo your change in 'cccp.c', or reinstall the original version, and do 'make cpp' again.
4. Copy this final version of GNU cpp into '/lib/cpp'.
5. Replace every occurrence of obstack_free in the file 'tree.c' with _obstack_free.
6. Run make to get the first-stage GNU CC.
7. Reinstall the original version of '/lib/cpp'.
8. Now you can compile GNU CC with itself and install it in the normal fashion.

'm68k-bull-sysv'
Bull DPX/2 series 200 and 300 with BOS-2.00.45 up to BOS-2.01. GNU CC works either with native assembler or GNU assembler. You can use GNU assembler with native coff generation by providing '--with-gnu-as' to the configure script or use GNU assembler with dbx-in-coff encapsulation by providing '--with-gnu-as --stabs'. For any problem with native assembler or for availability of the DPX/2 port of GAS, contact F.Pierresteguy@frcl.bull.fr.

'm68k-crds-unox'
Use 'configure unos' for building on Unos.

The Unos assembler is named casm instead of as. For some strange reason linking '/bin/as' to '/bin/casm' changes the behavior, and does not work. So, when installing GNU CC, you should install the following script as 'as' in the subdirectory where the passes of GCC are installed:

```
#!/bin/sh
casm $*
```

The default Unos library is named 'libunos.a' instead of 'libc.a'. To allow GNU CC to function, either change all references to '-lc' in 'gcc.c' to '-lunos' or link '/lib/libc.a' to '/lib/libunos.a'.

When compiling GNU CC with the standard compiler, to overcome bugs in the support of alloca, do not use '-O' when making stage 2. Then use the stage 2 compiler with '-O' to make the stage 3 compiler. This compiler will have the same characteristics as the usual stage 2 compiler on other

systems. Use it to make a stage 4 compiler and compare that with stage 3 to verify proper compilation.

(Perhaps simply defining `ALLOCA` in '`x-crds`' as described in the comments there will make the above paragraph superfluous. Please inform us of whether this works.)

Unos uses memory segmentation instead of demand paging, so you will need a lot of memory. 5 Mb is barely enough if no other tasks are running. If linking '`cc1`' fails, try putting the object files into a library and linking from that library.

'`m68k-hp-hpux`'
: HP 9000 series 300 or 400 running HP-UX. HP-UX version 8.0 has a bug in the assembler that prevents compilation of GNU CC. To fix it, get patch PHCO_4484 from HP.

  In addition, if you wish to use gas '`--with-gnu-as`' you must use gas version 2.1 or later, and you must use the GNU linker version 2.1 or later. Earlier versions of gas relied upon a program which converted the gas output into the native HP/UX format, but that program has not been kept up to date. gdb does not understand that native HP/UX format, so you must use gas if you wish to use gdb.

'`m68k-sun`'
: Sun 3. We do not provide a configuration file to use the Sun FPA by default, because programs that establish signal handlers for floating point traps inherently cannot work with the FPA.

  See Section 5.4 "Sun Install," page 133, for information on installing GNU CC on Sun systems.

'`m88k-*-svr3`'
: Motorola m88k running the AT&T/Unisoft/Motorola V.3 reference port. These systems tend to use the Green Hills C, revision 1.8.5, as the standard C compiler. There are apparently bugs in this compiler that result in object files differences between stage 2 and stage 3. If this happens, make the stage 4 compiler and compare it to the stage 3 compiler. If the stage 3 and stage 4 object files are identical, this suggests you encountered a problem with the standard C compiler; the stage 3 and 4 compilers may be usable.

  It is best, however, to use an older version of GNU CC for bootstrapping if you have one.

'`m88k-*-dgux`'
: Motorola m88k running DG/UX. To build 88open BCS native or cross compilers on DG/UX, specify the configuration

name as 'm88k-*-dguxbcs' and build in the 88open BCS software development environment. To build ELF native or cross compilers on DG/UX, specify 'm88k-*-dgux' and build in the DG/UX ELF development environment. You set the software development environment by issuing 'sde-target' command and specifying either 'm88kbcs' or 'm88kdguxelf' as the operand.

If you do not specify a configuration name, 'configure' guesses the configuration based on the current software development environment.

'm88k-tektronix-sysv3'
: Tektronix XD88 running UTekV 3.2e. Do not turn on optimization while building stage1 if you bootstrap with the buggy Green Hills compiler. Also, The bundled LAI System V NFS is buggy so if you build in an NFS mounted directory, start from a fresh reboot, or avoid NFS all together. Otherwise you may have trouble getting clean comparisons between stages.

'mips-mips-bsd'
: MIPS machines running the MIPS operating system in BSD mode. It's possible that some old versions of the system lack the functions memcpy, memcmp, and memset. If your system lacks these, you must remove or undo the definition of TARGET_MEM_FUNCTIONS in 'mips-bsd.h'.

The MIPS C compiler needs to be told to increase its table size for switch statements with the '-Wf,-XNg1500' option in order to compile 'cp/parse.c'. If you use the '-O2' optimization option, you also need to use '-Olimit 3000'. Both of these options are automatically generated in the 'Makefile' that the shell script 'configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add '-Wf,-XNg1500 -Olimit 3000'.

'mips-mips-riscos*'
: The MIPS C compiler needs to be told to increase its table size for switch statements with the '-Wf,-XNg1500' option in order to compile 'cp/parse.c'. If you use the '-O2' optimization option, you also need to use '-Olimit 3000'. Both of these options are automatically generated in the 'Makefile' that the shell script 'configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add '-Wf,-XNg1500 -Olimit 3000'.

MIPS computers running RISC-OS can support four different personalities: default, BSD 4.3, System V.3, and System

V.4 (older versions of RISC-OS don't support V.4). To configure GCC for these platforms use the following configurations:

'mips-mips-riscos*rev*'
: Default configuration for RISC-OS, revision *rev*.

'mips-mips-riscos*rev*bsd'
: BSD 4.3 configuration for RISC-OS, revision *rev*.

'mips-mips-riscos*rev*sysv4'
: System V.4 configuration for RISC-OS, revision *rev*.

'mips-mips-riscos*rev*sysv'
: System V.3 configuration for RISC-OS, revision *rev*.

The revision rev mentioned above is the revision of RISC-OS to use. You must reconfigure GCC when going from a RISC-OS revision 4 to RISC-OS revision 5. This has the effect of avoiding a linker bug (see Section 9.2 "Installation Problems," page 205, for more details).

'mips-sgi-*'

In order to compile GCC on an SGI running IRIX 4, the "c.hdr.lib" option must be installed from the CD-ROM supplied from Silicon Graphics. This is found on the 2nd CD in release 4.0.1.

In order to compile GCC on an SGI running IRIX 5, the "compiler_dev.hdr" subsystem must be installed from the IDO CD-ROM supplied by Silicon Graphics.

`make compare` may fail on version 5 of IRIX unless you add '`-save-temps`' to `CFLAGS`. On these systems, the name of the assembler input file is stored in the object file, and that makes comparison fail if it differs between the `stage1` and `stage2` compilations. The option '`-save-temps`' forces a fixed name to be used for the assembler input file, instead of a randomly chosen name in '`/tmp`'. Do not add '`-save-temps`' unless the comparisons fail without that option. If you do you '`-save-temps`', you will have to manually delete the '`.i`' and '`.s`' files after each series of compilations.

The MIPS C compiler needs to be told to increase its table size for switch statements with the '`-Wf,-XNg1500`' option in order to compile '`cp/parse.c`'. If you use the '`-O2`' optimization option, you also need to use '`-Olimit 3000`'. Both of these options are automatically generated in the '`Makefile`' that the shell script '`configure`' builds. If you override the `CC`

make variable and use the MIPS compilers, you may need to add '-Wf,-XNg1500 -Olimit 3000'.

On Irix version 4.0.5F, and perhaps on some other versions as well, there is an assembler bug that reorders instructions incorrectly. To work around it, specify the target configuration 'mips-sgi-irix4loser'. This configuration inhibits assembler optimization.

In a compiler configured with target 'mips-sgi-irix4', you can turn off assembler optimization by using the '-noasmopt' option. This compiler option passes the option '-O0' to the assembler, to inhibit reordering.

The '-noasmopt' option can be useful for testing whether a problem is due to erroneous assembler reordering. Even if a problem does not go away with '-noasmopt', it may still be due to assembler reordering—perhaps GNU CC itself was miscompiled as a result.

To enable debugging under Irix 5, you must use GNU as 2.5 or later, and use the '--with-gnu-as' configure option when configuring gcc. GNU as is distributed as part of the binutils package.

'mips-sony-sysv'
: Sony MIPS NEWS. This works in NEWSOS 5.0.1, but not in 5.0.2 (which uses ELF instead of COFF). Support for 5.0.2 will probably be provided soon by volunteers. In particular, the linker does not like the code generated by GCC when shared libraries are linked in.

'ns32k-encore'
: Encore ns32000 system. Encore systems are supported only under BSD.

'ns32k-*-genix'
: National Semiconductor ns32000 system. Genix has bugs in alloca and malloc; you must get the compiled versions of these from GNU Emacs.

'ns32k-sequent'
: Go to the Berkeley universe before compiling. In addition, you probably need to create a file named 'string.h' containing just one line: '#include <strings.h>'.

'ns32k-utek'
: UTEK ns32000 system ("merlin"). The C compiler that comes with this system cannot compile GNU CC; contact 'tektronix!reed!mason' to get binaries of GNU CC for bootstrapping.

'romp-*-aos'
'romp-*-mach'
: The only operating systems supported for the IBM RT PC are AOS and MACH. GNU CC does not support AIX running on the RT. We recommend you compile GNU CC with an earlier version of itself; if you compile GNU CC with `hc`, the Metaware compiler, it will work, but you will get mismatches between the stage 2 and stage 3 compilers in various files. These errors are minor differences in some floating-point constants and can be safely ignored; the stage 3 compiler is correct.

'rs6000-*-aix'
'powerpc-*-aix'
: Various early versions of each release of the IBM XLC compiler will not bootstrap GNU CC. Symptoms include differences between the stage2 and stage3 object files, and errors when compiling 'libgcc.a' or 'enquire'. Known problematic releases include: xlc-1.2.1.8, xlc-1.3.0.0 (distributed with AIX 3.2.5), and xlc-1.3.0.19. Both xlc-1.2.1.28 and xlc-1.3.0.24 (PTF 432238) are known to produce working versions of GNU CC, but most other recent releases correctly bootstrap GNU CC. Also, releases of AIX prior to AIX 3.2.4 include a version of the IBM assembler which does not accept debugging directives: assembler updates are available as PTFs. See the file 'README.RS6000' for more details on both of these problems.

  Only AIX is supported on the PowerPC. GNU CC does not yet support the 64-bit PowerPC instructions.

  Objective C does not work on this architecture.

  AIX on the RS/6000 provides support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ("." vs "," for separating decimal fractions). There have been problems reported where the library linked with GNU CC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the LANG environment variable to "C" or "En_US".

'powerpc-*-elf'
'powerpc-*-eabi'
'powerpc-*-sysv4'
: These systems are currently under development.

'vax-dec-ultrix'

Don't try compiling with Vax C (vcc). It produces incorrect code in some cases (for example, when alloca is used).

Meanwhile, compiling 'cp/parse.c' with pcc does not work because of an internal table size limitation in that compiler. To avoid this problem, compile just the GNU C compiler first, and use it to recompile building all the languages that you want to run.

'sparc-sun-*'

See Section 5.4 "Sun Install," page 133, for information on installing GNU CC on Sun systems.

'vax-dec-vms'

See Section 5.5 "VMS Install," page 133, for details on how to install GNU CC on VMS.

'we32k-*-*'

These computers are also known as the 3b2, 3b5, 3b20 and other similar names. (However, the 3b1 is actually a 68000; see Section 5.1 "Configurations," page 111.)

Don't use '-g' when compiling with the system's compiler. The system's linker seems to be unable to handle such a large program with debugging information.

The system's compiler runs out of capacity when compiling 'stmt.c' in GNU CC. You can work around this by building 'cpp' in GNU CC first, then use that instead of the system's preprocessor with the system's C compiler to compile 'stmt.c'. Here is how:

```
mv /lib/cpp /lib/cpp.att
cp cpp /lib/cpp.gnu
echo '/lib/cpp.gnu -traditional ${1+"$@"}' > /lib/cpp
chmod +x /lib/cpp
```

The system's compiler produces bad code for some of the GNU CC optimization files. So you must build the stage 2 compiler without optimization. Then build a stage 3 compiler with optimization. That executable should work. Here are the necessary commands:

```
make LANGUAGES=c CC=stage1/xgcc CFLAGS="-Bstage1/ -g"
make stage2
make CC=stage2/xgcc CFLAGS="-Bstage2/ -g -O"
```

You may need to raise the ULIMIT setting to build a C++ compiler, as the file 'cc1plus' is larger than one megabyte.

5.2 Compilation in a Separate Directory

If you wish to build the object files and executables in a directory other than the one containing the source files, here is what you must do differently:

1. Make sure you have a version of Make that supports the `VPATH` feature. (GNU Make supports it, as do Make versions on most BSD systems.)
2. If you have ever run '`configure`' in the source directory, you must undo the configuration. Do this by running:
   ```
   make distclean
   ```
3. Go to the directory in which you want to build the compiler before running '`configure`':
   ```
   mkdir gcc-sun3
   cd gcc-sun3
   ```
   On systems that do not support symbolic links, this directory must be on the same file system as the source code directory.
4. Specify where to find '`configure`' when you run it:
   ```
   ../gcc/configure ...
   ```
   This also tells `configure` where to find the compiler sources; `configure` takes the directory from the file name that was used to invoke it. But if you want to be sure, you can specify the source directory with the '`--srcdir`' option, like this:
   ```
   ../gcc/configure --srcdir=../gcc other options
   ```
   The directory you specify with '`--srcdir`' need not be the same as the one that `configure` is found in.

Now, you can run `make` in that directory. You need not repeat the configuration steps shown above, when ordinary source files change. You must, however, run `configure` again when the configuration files change, if your system does not support symbolic links.

5.3 Building and Installing a Cross-Compiler

GNU CC can function as a cross-compiler for many machines, but not all.

- Cross-compilers for the Mips as target using the Mips assembler currently do not work, because the auxiliary programs '`mips-tdump.c`' and '`mips-tfile.c`' can't be compiled on anything but a Mips. It does work to cross compile for a Mips if you use the GNU assembler and linker.
- Cross-compilers between machines with different floating point formats have not all been made to work. GNU CC now has a floating

point emulator with which these can work, but each target machine description needs to be updated to take advantage of it.

- Cross-compilation between machines of different word sizes is somewhat problematic and sometimes does not work.

Since GNU CC generates assembler code, you probably need a cross-assembler that GNU CC can run, in order to produce object files. If you want to link on other than the target machine, you need a cross-linker as well. You also need header files and libraries suitable for the target machine that you can install on the host machine.

5.3.1 Steps of Cross-Compilation

To compile and run a program using a cross-compiler involves several steps:

- Run the cross-compiler on the host machine to produce assembler files for the target machine. This requires header files for the target machine.
- Assemble the files produced by the cross-compiler. You can do this either with an assembler on the target machine, or with a cross-assembler on the host machine.
- Link those files to make an executable. You can do this either with a linker on the target machine, or with a cross-linker on the host machine. Whichever machine you use, you need libraries and certain startup files (typically '`crt. . . .o`') for the target machine.

It is most convenient to do all of these steps on the same host machine, since then you can do it all with a single invocation of GNU CC. This requires a suitable cross-assembler and cross-linker. For some targets, the GNU assembler and linker are available.

5.3.2 Configuring a Cross-Compiler

To build GNU CC as a cross-compiler, you start out by running '`configure`'. Use the '`--target=`*target*' to specify the target type. If '`configure`' was unable to correctly identify the system you are running on, also specify the '`--build=`*build*' option. For example, here is how to configure for a cross-compiler that produces code for an HP 68030 system running BSD on a system that '`configure`' can correctly identify:

```
./configure --target=m68k-hp-bsd4.3
```

5.3.3 Tools and Libraries for a Cross-Compiler

If you have a cross-assembler and cross-linker available, you should install them now. Put them in the directory '/usr/local/*target*/bin'. Here is a table of the tools you should put in this directory:

'as' This should be the cross-assembler.

'ld' This should be the cross-linker.

'ar' This should be the cross-archiver: a program which can manipulate archive files (linker libraries) in the target machine's format.

'ranlib' This should be a program to construct a symbol table in an archive file.

The installation of GNU CC will find these programs in that directory, and copy or link them to the proper place to for the cross-compiler to find them when run later.

The easiest way to provide these files is to build the Binutils package and GAS. Configure them with the same '--host' and '--target' options that you use for configuring GNU CC, then build and install them. They install their executables automatically into the proper directory. Alas, they do not support all the targets that GNU CC supports.

If you want to install libraries to use with the cross-compiler, such as a standard C library, put them in the directory '/usr/local/*target*/lib'; installation of GNU CC copies all all the files in that subdirectory into the proper place for GNU CC to find them and link with them. Here's an example of copying some libraries from a target machine:

```
ftp target-machine
lcd /usr/local/target/lib
cd /lib
get libc.a
cd /usr/lib
get libg.a
get libm.a
quit
```

The precise set of libraries you'll need, and their locations on the target machine, vary depending on its operating system.

Many targets require "start files" such as 'crt0.o' and 'crtn.o' which are linked into each executable; these too should be placed in '/usr/local/*target*/lib'. There may be several alternatives for 'crt0.o', for use with profiling or other compilation options. Check your target's definition of STARTFILE_SPEC to find out what start files it uses. Here's an example of copying these files from a target machine:

```
ftp target-machine
```

```
lcd /usr/local/target/lib
prompt
cd /lib
mget *crt*.o
cd /usr/lib
mget *crt*.o
quit
```

5.3.4 'libgcc.a' and Cross-Compilers

Code compiled by GNU CC uses certain runtime support functions implicitly. Some of these functions can be compiled successfully with GNU CC itself, but a few cannot be. These problem functions are in the source file 'libgcc1.c'; the library made from them is called 'libgcc1.a'.

When you build a native compiler, these functions are compiled with some other compiler–the one that you use for bootstrapping GNU CC. Presumably it knows how to open code these operations, or else knows how to call the run-time emulation facilities that the machine comes with. But this approach doesn't work for building a cross-compiler. The compiler that you use for building knows about the host system, not the target system.

So, when you build a cross-compiler you have to supply a suitable library 'libgcc1.a' that does the job it is expected to do.

To compile 'libgcc1.c' with the cross-compiler itself does not work. The functions in this file are supposed to implement arithmetic operations that GNU CC does not know how to open code, for your target machine. If these functions are compiled with GNU CC itself, they will compile into infinite recursion.

On any given target, most of these functions are not needed. If GNU CC can open code an arithmetic operation, it will not call these functions to perform the operation. It is possible that on your target machine, none of these functions is needed. If so, you can supply an empty library as 'libgcc1.a'.

Many targets need library support only for multiplication and division. If you are linking with a library that contains functions for multiplication and division, you can tell GNU CC to call them directly by defining the macros `MULSI3_LIBCALL`, and the like. These macros need to be defined in the target description macro file. For some targets, they are defined already. This may be sufficient to avoid the need for libgcc1.a; if so, you can supply an empty library.

Some targets do not have floating point instructions; they need other functions in 'libgcc1.a', which do floating arithmetic. Recent versions of GNU CC have a file which emulates floating point. With a certain amount of work, you should be able to construct a floating point emulator

that can be used as 'libgcc1.a'. Perhaps future versions will contain code to do this automatically and conveniently. That depends on whether someone wants to implement it.

If your target system has another C compiler, you can configure GNU CC as a native compiler on that machine, build just 'libgcc1.a' with 'make libgcc1.a' on that machine, and use the resulting file with the cross-compiler. To do this, execute the following on the target machine:

```
cd target-build-dir
./configure --host=sparc --target=sun3
make libgcc1.a
```

And then this on the host machine:

```
ftp target-machine
binary
cd target-build-dir
get libgcc1.a
quit
```

Another way to provide the functions you need in 'libgcc1.a' is to define the appropriate perform_... macros for those functions. If these definitions do not use the C arithmetic operators that they are meant to implement, you should be able to compile them with the cross-compiler you are building. (If these definitions already exist for your target file, then you are all set.)

To build 'libgcc1.a' using the perform macros, build the compiler using 'LIBGCC1=libgcc1.a OLDCC=./xgcc'. Otherwise, you should place your replacement library under the name 'libgcc1.a' in the directory in which you will build the cross-compiler, before you run make.

5.3.5 Cross-Compilers and Header Files

If you are cross-compiling a standalone program or a program for an embedded system, then you may not need any header files except the few that are part of GNU CC (and those of your program). However, if you intend to link your program with a standard C library such as 'libc.a', then you probably need to compile with the header files that go with the library you use.

The GNU C compiler does not come with these files, because (1) they are system-specific, and (2) they belong in a C library, not in a compiler.

If the GNU C library supports your target machine, then you can get the header files from there (assuming you actually use the GNU library when you link your program).

If your target machine comes with a C compiler, it probably comes with suitable header files also. If you make these files accessible from the host machine, the cross-compiler can use them also.

Otherwise, you're on your own in finding header files to use when cross-compiling.

When you have found suitable header files, put them in '/usr/local/*target*/include', before building the cross compiler. Then installation will run fixincludes properly and install the corrected versions of the header files where the compiler will use them.

Provide the header files before you build the cross-compiler, because the build stage actually runs the cross-compiler to produce parts of 'libgcc.a'. (These are the parts that *can* be compiled with GNU CC.) Some of them need suitable header files.

Here's an example showing how to copy the header files from a target machine. On the target machine, do this:

```
(cd /usr/include; tar cf - .) > tarfile
```

Then, on the host machine, do this:

```
ftp target-machine
lcd /usr/local/target/include
get tarfile
quit
tar xf tarfile
```

5.3.6 Actually Building the Cross-Compiler

Now you can proceed just as for compiling a single-machine compiler through the step of building stage 1. If you have not provided some sort of 'libgcc1.a', then compilation will give up at the point where it needs that file, printing a suitable error message. If you do provide 'libgcc1.a', then building the compiler will automatically compile and link a test program called 'cross-test'; if you get errors in the linking, it means that not all of the necessary routines in 'libgcc1.a' are available.

If you are making a cross-compiler for an embedded system, and there is no 'stdio.h' header for it, then the compilation of 'enquire' will probably fail. The job of 'enquire' is to run on the target machine and figure out by experiment the nature of its floating point representation. 'enquire' records its findings in the header file 'float.h'. If you can't produce this file by running 'enquire' on the target machine, then you will need to come up with a suitable 'float.h' in some other way (or else, avoid using it in your programs).

Do not try to build stage 2 for a cross-compiler. It doesn't work to rebuild GNU CC as a cross-compiler using the cross-compiler, because that would produce a program that runs on the target machine, not on the host. For example, if you compile a 386-to-68030 cross-compiler with itself, the result will not be right either for the 386 (because it was compiled into 68030 code) or for the 68030 (because it was configured

for a 386 as the host). If you want to compile GNU CC into 68030 code, whether you compile it on a 68030 or with a cross-compiler on a 386, you must specify a 68030 as the host when you configure it.

To install the cross-compiler, use 'make install', as usual.

5.4 Installing GNU CC on the Sun

On Solaris (version 2.1), do not use the linker or other tools in '/usr/ucb' to build GNU CC. Use /usr/ccs/bin.

Make sure the environment variable FLOAT_OPTION is not set when you compile 'libgcc.a'. If this option were set to f68881 when 'libgcc.a' is compiled, the resulting code would demand to be linked with a special startup file and would not link properly without special pains.

There is a bug in alloca in certain versions of the Sun library. To avoid this bug, install the binaries of GNU CC that were compiled by GNU CC. They use alloca as a built-in function and never the one in the library.

Some versions of the Sun compiler crash when compiling GNU CC. The problem is a segmentation fault in cpp. This problem seems to be due to the bulk of data in the environment variables. You may be able to avoid it by using the following command to compile GNU CC with Sun CC:

```
make CC="TERMCAP=x OBJS=x LIBFUNCS=x STAGESTUFF=x cc"
```

5.5 Installing GNU CC on VMS

The VMS version of GNU CC is distributed in a backup saveset containing both source code and precompiled binaries.

To install the 'gcc' command so you can use the compiler easily, in the same manner as you use the VMS C compiler, you must install the VMS CLD file for GNU CC as follows:

1. Define the VMS logical names 'GNU_CC' and 'GNU_CC_INCLUDE' to point to the directories where the GNU CC executables ('gcc-cpp.exe', 'gcc-cc1.exe', etc.) and the C include files are kept respectively. This should be done with the commands:

   ```
   $ assign /system /translation=concealed -
     disk:[gcc.] gnu_cc
   $ assign /system /translation=concealed -
     disk:[gcc.include.] gnu_cc_include
   ```

   with the appropriate disk and directory names. These commands can be placed in your system startup file so they will be executed

whenever the machine is rebooted. You may, if you choose, do this via the 'GCC_INSTALL.COM' script in the '[GCC]' directory.

2. Install the 'GCC' command with the command line:

```
$ set command /table=sys$common:[syslib]dcltables -
  /output=sys$common:[syslib]dcltables gnu_cc:[000000]gcc
$ install replace sys$common:[syslib]dcltables
```

3. To install the help file, do the following:

```
$ library/help sys$library:helplib.hlb gcc.hlp
```

   Now you can invoke the compiler with a command like 'gcc /verbose file.c', which is equivalent to the command 'gcc -v -c file.c' in Unix.

If you wish to use GNU C++ you must first install GNU CC, and then perform the following steps:

1. Define the VMS logical name 'GNU_GXX_INCLUDE' to point to the directory where the preprocessor will search for the C++ header files. This can be done with the command:

```
$ assign /system /translation=concealed -
  disk:[gcc.gxx_include.] gnu_gxx_include
```

   with the appropriate disk and directory name. If you are going to be using libg++, this is where the libg++ install procedure will install the libg++ header files.

2. Obtain the file 'gcc-cc1plus.exe', and place this in the same directory that 'gcc-cc1.exe' is kept.

   The GNU C++ compiler can be invoked with a command like 'gcc /plus /verbose file.cc', which is equivalent to the command 'g++ -v -c file.cc' in Unix.

We try to put corresponding binaries and sources on the VMS distribution tape. But sometimes the binaries will be from an older version than the sources, because we don't always have time to update them. (Use the '/version' option to determine the version number of the binaries and compare it with the source file 'version.c' to tell whether this is so.) In this case, you should use the binaries you get to recompile the sources. If you must recompile, here is how:

1. Execute the command procedure 'vmsconfig.com' to set up the files 'tm.h', 'config.h', 'aux-output.c', and 'md.', and to create files 'tconfig.h' and 'hconfig.h'. This procedure also creates several linker option files used by 'make-cc1.com' and a data file used by 'make-l2.com'.

```
$ @vmsconfig.com
```

2. Setup the logical names and command tables as defined above. In addition, define the VMS logical name 'GNU_BISON' to point at the to

the directories where the Bison executable is kept. This should be done with the command:

```
$ assign /system /translation=concealed -
  disk:[bison.] gnu_bison
```

You may, if you choose, use the 'INSTALL_BISON.COM' script in the '[BISON]' directory.

3. Install the 'BISON' command with the command line:

```
$ set command /table=sys$common:[syslib]dcltables -
  /output=sys$common:[syslib]dcltables -
  gnu_bison:[000000]bison
$ install replace sys$common:[syslib]dcltables
```

4. Type '@make-gcc' to recompile everything (alternatively, submit the file 'make-gcc.com' to a batch queue). If you wish to build the GNU C++ compiler as well as the GNU CC compiler, you must first edit 'make-gcc.com' and follow the instructions that appear in the comments.
5. In order to use GCC, you need a library of functions which GCC compiled code will call to perform certain tasks, and these functions are defined in the file 'libgcc2.c'. To compile this you should use the command procedure 'make-l2.com', which will generate the library 'libgcc2.olb'. 'libgcc2.olb' should be built using the compiler built from the same distribution that 'libgcc2.c' came from, and 'make-gcc.com' will automatically do all of this for you.

   To install the library, use the following commands:

```
$ library gnu_cc:[000000]gcclib/delete=(new,eprintf)
$ library gnu_cc:[000000]gcclib/delete=L_*
$ library libgcc2/extract=*/output=libgcc2.obj
$ library gnu_cc:[000000]gcclib libgcc2.obj
```

   The first command simply removes old modules that will be replaced with modules from 'libgcc2' under different module names. The modules new and eprintf may not actually be present in your 'gcclib.olb'—if the VMS librarian complains about those modules not being present, simply ignore the message and continue on with the next command. The second command removes the modules that came from the previous version of the library 'libgcc2.c'.

   Whenever you update the compiler on your system, you should also update the library with the above procedure.
6. You may wish to build GCC in such a way that no files are written to the directory where the source files reside. An example would be the when the source files are on a read-only disk. In these cases, execute the following DCL commands (substituting your actual path names):

```
$ assign dua0:[gcc.build_dir.]/translation=concealed, -
         dua1:[gcc.source_dir.]/translation=concealed
                          gcc_build
```

```
$ set default gcc_build:[000000]
```

where the directory 'dua1:[gcc.source_dir]' contains the source code, and the directory 'dua0:[gcc.build_dir]' is meant to contain all of the generated object files and executables. Once you have done this, you can proceed building GCC as described above. (Keep in mind that 'gcc_build' is a rooted logical name, and thus the device names in each element of the search list must be an actual physical device name rather than another rooted logical name).

7. **If you are building GNU CC with a previous version of GNU CC, you also should check to see that you have the newest version of the assembler**. In particular, GNU CC version 2 treats global constant variables slightly differently from GNU CC version 1, and GAS version 1.38.1 does not have the patches required to work with GCC version 2. If you use GAS 1.38.1, then `extern const` variables will not have the read-only bit set, and the linker will generate warning messages about mismatched psect attributes for these variables. These warning messages are merely a nuisance, and can safely be ignored.

   If you are compiling with a version of GNU CC older than 1.33, specify '/DEFINE=("inline=")' as an option in all the compilations. This requires editing all the `gcc` commands in 'make-cc1.com'. (The older versions had problems supporting `inline`.) Once you have a working 1.33 or newer GNU CC, you can change this file back.

8. If you want to build GNU CC with the VAX C compiler, you will need to make minor changes in 'make-cccp.com' and 'make-cc1.com' to choose alternate definitions of `CC`, `CFLAGS`, and `LIBS`. See comments in those files. However, you must also have a working version of the GNU assembler (GNU as, aka GAS) as it is used as the back-end for GNU CC to produce binary object modules and is not included in the GNU CC sources. GAS is also needed to compile 'libgcc2' in order to build 'gcclib' (see above); 'make-l2.com' expects to be able to find it operational in 'gnu_cc:[000000]gnu-as.exe'.

   To use GNU CC on VMS, you need the VMS driver programs 'gcc.exe', 'gcc.com', and 'gcc.cld'. They are distributed with the VMS binaries ('gcc-vms') rather than the GNU CC sources. GAS is also included in 'gcc-vms', as is Bison.

   Once you have successfully built GNU CC with VAX C, you should use the resulting compiler to rebuild itself. Before doing this, be sure to restore the `CC`, `CFLAGS`, and `LIBS` definitions in 'make-cccp.com' and 'make-cc1.com'. The second generation compiler will be able to take advantage of many optimizations that must be suppressed when building with other compilers.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked with the sharable 'VAXCRTL' library. Now this should work.

Even with this version, however, GNU CC itself should not be linked with the sharable 'VAXCRTL'. The version of qsort in 'VAXCRTL' has a bug (known to be present in VMS versions V4.6 through V5.5) which causes the compiler to fail.

The executables are generated by 'make-cc1.com' and 'make-cccp.com'. Use the object library version of 'VAXCRTL' in order to make use of the qsort routine in 'gcclib.olb'. If you wish to link the compiler executables with the shareable image version of 'VAXCRTL', you should edit the file 'tm.h' (created by 'vmsconfig.com') to define the macro QSORT_WORKAROUND.

QSORT_WORKAROUND is always defined when GNU CC is compiled with VAX C, to avoid a problem in case 'gcclib.olb' is not yet available.

5.6 collect2

Many target systems do not have support in the assembler and linker for "constructors"—initialization functions to be called before the official "start" of main. On such systems, GNU CC uses a utility called collect2 to arrange to call these functions at start time.

The program collect2 works by linking the program once and looking through the linker output file for symbols with particular names indicating they are constructor functions. If it finds any, it creates a new temporary '.c' file containing a table of them, compiles it, and links the program a second time including that file.

The actual calls to the constructors are carried out by a subroutine called __main, which is called (automatically) at the beginning of the body of main (provided main was compiled with GNU CC). Calling __main is necessary, even when compiling C code, to allow linking C and C++ object code together. (If you use '-nostdlib', you get an unresolved reference to __main, since it's defined in the standard GCC library. Include '-lgcc' at the end of your compiler command line to resolve this reference.)

The program collect2 is installed as ld in the directory where the passes of the compiler are installed. When collect2 needs to find the *real* ld, it tries the following file names:

- 'real-ld' in the directories listed in the compiler's search directories.
- 'real-ld' in the directories listed in the environment variable PATH.
- The file specified in the REAL_LD_FILE_NAME configuration macro, if specified.

- 'ld' in the compiler's search directories, except that collect2 will not execute itself recursively.
- 'ld' in PATH.

"The compiler's search directories" means all the directories where gcc searches for passes of the compiler. This includes directories that you specify with '-B'.

Cross-compilers search a little differently:

- 'real-ld' in the compiler's search directories.
- '*target*-real-ld' in PATH.
- The file specified in the REAL_LD_FILE_NAME configuration macro, if specified.
- 'ld' in the compiler's search directories.
- '*target*-ld' in PATH.

collect2 explicitly avoids running ld using the file name under which collect2 itself was invoked. In fact, it remembers up a list of such names—in case one copy of collect2 finds another copy (or version) of collect2 installed as ld in a second place in the search path.

collect2 searches for the utilities nm and strip using the same algorithm as above for ld.

5.7 Standard Header File Directories

GCC_INCLUDE_DIR means the same thing for native and cross. It is where GNU CC stores its private include files, and also where GNU CC stores the fixed include files. A cross compiled GNU CC runs fixincludes on the header files in '$(tooldir)/include'. (If the cross compilation header files need to be fixed, they must be installed before GNU CC is built. If the cross compilation header files are already suitable for ANSI C and GNU CC, nothing special need be done).

GPLUS_INCLUDE_DIR means the same thing for native and cross. It is where g++ looks first for header files. libg++ installs only target independent header files in that directory.

LOCAL_INCLUDE_DIR is used only for a native compiler. It is normally '/usr/local/include'. GNU CC searches this directory so that users can install header files in '/usr/local/include'.

CROSS_INCLUDE_DIR is used only for a cross compiler. GNU CC doesn't install anything there.

TOOL_INCLUDE_DIR is used for both native and cross compilers. It is the place for other packages to install header files that GNU CC will use. For a cross-compiler, this is the equivalent of '/usr/include'. When you

build a cross-compiler, `fixincludes` processes any header files in this directory.

6 Extensions to the C Language Family

GNU C provides several language features not found in ANSI standard C. (The '-pedantic' option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro __GNUC__, which is always defined under GNU CC.

These extensions are available in C and Objective C. Most of them are also available in C++. See Chapter 7 "Extensions to the C++ Language," page 189, for extensions that apply *only* to C++.

6.1 Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of foo ().

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type void, and thus effectively no value.)

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either *a* or *b* twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let's assume int), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see Section 6.7 "Typeof," page 147) or type naming (see Section 6.6 "Naming Types," page 146).

6.2 Locally Declared Labels

Each statement expression is a scope in which *local labels* can be declared. A local label is simply an identifier; you can jump to it with an ordinary `goto` statement, but only from within the statement expression it belongs to.

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, ...;
```

Local label declarations must come at the beginning of the statement expression, right after the '`({`', before any ordinary declarations.

The label declaration defines the label *name*, but does not define the label itself. You must do this in the usual way, with `label:`, within the statements of the statement expression.

The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a `goto` can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(array, target)                     \
({                                                \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { value = i; goto found; }                \
  value = -1;                                     \
 found:                                           \
  value;                                          \
})
```

6.3 Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator '`&&`'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement[1], `goto *exp;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds—array indexing in C never does that.

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use that rather than an array unless the problem does not fit a `switch` statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You can use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

6.4 Nested Functions

A *nested function* is a function defined inside another function. (Nested functions are not supported for GNU C++.) The nested function's name is local to the block where it is defined. For example, here we define a nested function named `square`, and call it twice:

[1] The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

```
foo (double a, double b)
{
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. For example, here we show a nested function which uses an inherited variable named `offset`:

```
bar (int *array, int offset, int size)
{
  int access (int *array, int index)
    { return array[index + offset]; }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
}
```

Nested function definitions are permitted within functions in the places where variable definitions are allowed; that is, in any block, before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
hack (int *array, int size)
{
  void store (int index, int value)
    { array[index] = value; }

  intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. But this technique works only so long as the containing function (`hack`, in this example) does not exit.

If you try to call the nested function through its address after the containing function has exited, all hell will break loose. If you try to call it after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

GNU CC implements taking the address of a nested function using a technique called *trampolines*. A paper describing them is available from '`maya.idiap.ch`' in directory '`pub/tmb`', file '`usenix88-lexic.ps.Z`'.

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (see Section 6.2 “Local Labels,” page 142). Such a jump returns instantly to the containing function, exiting the nested function which did the `goto` and any intermediate functions as well. Here is an example:

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
  ...
  return 0;

 /* Control comes here from access
    if it detects an error.  */
 failure:
  return -1;
}
```

A nested function always has internal linkage. Declaring one with `extern` is erroneous. If you need to declare the nested function before its definition, use `auto` (which is otherwise meaningless for function declarations).

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  auto int access (int *, int);
  ...
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  ...
}
```

6.5 Constructing Function Calls

Using the built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

`__builtin_apply_args ()`
: This built-in function returns a pointer of type `void *` to data describing how to perform a call with the same arguments as were passed to the current function.

  The function saves the arg pointer register, structure value address, and all registers that might be used to pass arguments to a function into a block of memory allocated on the stack. Then it returns the address of that block.

`__builtin_apply (`*`function`*`, `*`arguments`*`, `*`size`*`)`
: This built-in function invokes *`function`* (type `void (*)()`) with a copy of the parameters described by *`arguments`* (type `void *`) and *`size`* (type `int`).

  The value of *`arguments`* should be the value returned by `__builtin_apply_args`. The argument *`size`* specifies the size of the stack argument data, in bytes.

  This function returns a pointer of type `void *` to data describing how to return whatever value was returned by *`function`*. The data is saved in a block of memory allocated on the stack.

  It is not always simple to compute the proper value for *`size`*. The value is used by `__builtin_apply` to compute the amount of data that should be pushed on the stack and copied from the incoming argument area.

`__builtin_return (`*`result`*`)`
: This built-in function returns the value described by *`result`* from the containing function. You should specify, for *`result`*, a value returned by `__builtin_apply`.

6.6 Naming an Expression's Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define *`name`* as a type name for the type of *`exp`*:

```
typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a,b) \
  ({typedef _ta = (a), _tb = (b);  \
    _ta _a = (a); _tb _b = (b);     \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

6.7 Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of `typeof`. See Section 6.35 "Alternate Keywords," page 187.

A `typeof`-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

  ```
  typeof (*x) y;
  ```

- This declares `y` as an array of such values.

  ```
  typeof (*x) y[4];
  ```

- This declares `y` as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

  It is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

```
#define pointer(T)  typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, `array (pointer (char), 4)` is the type of arrays of 4 pointers to `char`.

6.8 Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

Standard C++ allows compound expressions and conditional expressions as lvalues, and permits casts to reference type, so use of this extension is deprecated for C++ code.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is an lvalue. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if `a` has type `char *`, the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)(int)5)
```

An assignment-with-arithmetic operation such as '`+=`' applied to a cast performs the arithmetic using the type resulting from the cast, and

then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *)(int) ((int)a + 5))
```

You cannot take the address of an lvalue cast, because the use of its address would not work out coherently. Suppose that `&(int)f` were permitted, where `f` has type `float`. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

```
*&(int)f = 1;
```

This is quite different from what `(int)f = 1` would do—that would convert 1 to floating point and store it. Rather than cause this inconsistency, we think it is better to prohibit use of '`&`' on a cast.

If you really do want an `int *` pointer with the address of `f`, you can simply write `(int *)&f`.

6.9 Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

6.10 Double-Word Integers

GNU C supports data types for integers that are twice as long as `long int`. Simply write `long long int` for a signed integer, or `unsigned long long int` for an unsigned integer. To make an integer constant of type `long long int`, add the suffix `LL` to the integer. To make an integer constant of type `unsigned long long int`, add the suffix `ULL` to the integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bitwise boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports fullword-to-doubleword a widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use special library routines that come with GNU CC.

There may be pitfalls when you use `long long` types for function arguments, unless you declare function prototypes. If a function expects type `int` for its argument, and you pass a value of type `long long int`, confusion will result because the caller and the subroutine will disagree about the number of bytes for the argument. Likewise, if the function expects `long long int` and you pass `int`. The best way to avoid such problems is to use prototypes.

6.11 Complex Numbers

GNU C supports complex data types. You can declare both complex integer types and complex floating types, using the keyword `__complex__`.

For example, '`__complex__ double x;`' declares `x` as a variable whose real part and imaginary part are both of type `double`. '`__complex__ short int y;`' declares `y` to have real and imaginary parts of type `short int`; this is not likely to be useful, but it shows that the set of complex types is complete.

To write a constant with a complex data type, use the suffix '`i`' or '`j`' (either one; they are equivalent). For example, `2.5fi` has type `__complex__ float` and `3i` has type `__complex__ int`. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant.

To extract the real part of a complex-valued expression *exp*, write `__real__` *exp*. Likewise, use `__imag__` to extract the imaginary part.

The operator '`~`' performs complex conjugation when used on a value with a complex type.

GNU CC can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice-versa). None of the supported debugging info formats has a way to represent noncontiguous allocation like this, so GNU CC describes a noncontiguous complex variable as if it were two separate variables of noncomplex type. If the variable's actual name is `foo`, the two fictitious variables are named `foo$real` and

`foo$imag`. You can examine and set these two fictitious variables with your debugger.

A future version of GDB will know how to recognize such pairs and treat them as a single variable with a complex type.

6.12 Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline = (struct line *)
    malloc (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

6.13 Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca`.)

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len][len])
{
  ...
}
```

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case you access it with `sizeof`.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list—another GNU extension.

```
struct entry
tester (int len; char data[len][len], int len)
{
  ...
}
```

The '`int len`' before the semicolon is a *parameter forward declaration*, and it serves the purpose of making the name `len` known when the declaration of `data` is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type.

6.14 Macros with Variable Numbers of Arguments

In GNU C, a macro can accept a variable number of arguments, much as a function can. The syntax for defining the macro looks much like that used for a function. Here is an example:

```
#define eprintf(format, args...)  \
 fprintf (stderr, format , ## args)
```

Here `args` is a *rest argument*: it takes in zero or more arguments, as many as the call contains. All of them plus the commas between them form the value of `args`, which is substituted into the macro body where `args` is used. Thus, we have this expansion:

```
eprintf ("%s:%d: ", input_file_name, line_number)
↦
fprintf (stderr, "%s:%d: " , input_file_name, line_number)
```

Note that the comma after the string constant comes from the definition of `eprintf`, whereas the last comma comes from the value of `args`.

The reason for using '`##`' is to handle the case when `args` matches no arguments at all. In this case, `args` has an empty value. In this case, the second comma in the definition becomes an embarrassment: if it got through to the expansion of the macro, we would get something like this:

```
fprintf (stderr, "success!\n" , )
```

which is invalid C syntax. '`##`' gets rid of the comma, so we get the following instead:

```
fprintf (stderr, "success!\n")
```

This is a special feature of the GNU C preprocessor: '`##`' before a rest argument that is empty discards the preceding sequence of non-whitespace characters from the macro definition. (If another macro argument precedes, none of it is discarded.)

It might be better to discard the last preprocessor token instead of the last preceding sequence of non-whitespace characters; in fact, we may someday change this feature to do so. We advise you to write the macro definition so that the preceding sequence of non-whitespace characters is just a single token, so that the meaning will not change if we change the definition of this feature.

6.15 Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary '`&`' operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {int a[4];};

struct foo f();

bar (int index)
{
  return f().a[index];
}
```

6.16 Arithmetic on `void`- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

The option '`-Wpointer-arith`' requests a warning if these extensions are used.

6.17 Non-Constant Initializers

As in standard C++, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
}
```

6.18 Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer.

Usually, the specified type is a structure. Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a `struct foo` with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two

valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a `switch` statement, while the latter does the same thing an ordinary C initializer would do. Here is an example of subscripting an array constructor:

```
output = ((int[]) { 2, x, 28 }) [input];
```

Constructor expressions for scalar types and union types are is also allowed, but then the constructor expression is equivalent to a cast.

6.19 Labeled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In GNU C you can give the elements in any order, specifying the array indices or structure field names they apply to. This extension is not implemented in GNU C++.

To specify an array index, write '`[`*index*`]`' or '`[`*index*`] =`' before the element value. For example,

```
int a[6] = { [4] 29, [2] = 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

To initialize a range of elements to the same value, write '`[`*first* `...` *last*`] =` *value*'. For example,

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

Note that the length of the array is the highest value specified plus one.

In a structure initializer, specify the name of a field to initialize with '*fieldname*`:`' before the element value. For example, given the following structure,

```
struct point { int x, y; };
```

the following initialization

```
struct point p = { y: yvalue, x: xvalue };
```

is equivalent to

```
struct point p = { xvalue, yvalue };
```

Another syntax which has the same meaning is '`.`*fieldname* `=`'., as shown here:

```
struct point p = { .y = yvalue, .x = xvalue };
```

You can also use an element label (with either the colon syntax or the period-equal syntax) when initializing a union, to specify which element of the union should be used. For example,

```
union foo { int i; double d; };

union foo f = { d: 4 };
```

will convert 4 to a `double` to store it in the union using the second element. By contrast, casting 4 to type `union foo` would store it into the union as the integer `i`, since it is an integer. (See Section 6.21 “Cast to Union,” page 157.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a label applies to the next consecutive element of the array or structure. For example,

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type. For example:

```
int whitespace[256]
  = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
      ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

6.20 Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from *`low`* to *`high`*, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the `...`, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

6.21 Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with `union tag` or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See Section 6.18 "Constructors," page 154.)

The types that may be cast to the union type are those of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

both `x` and `y` can be cast to type `union` foo.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
. . .
u = (union foo) x  ≡  u.i = x
u = (union foo) y  ≡  u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
. . .
hack ((union foo) x);
```

6.22 Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. Six attributes, `noreturn`, `const`, `format`, `section`, `constructor`, and `destructor` are currently defined for functions. Other attributes, including `section` are supported for variables declarations (see Section 6.28 "Variable Attributes," page 163) and for types (see Section 6.29 "Type Attributes," page 166)..

You may also specify attributes with '__' preceeding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__noreturn__` instead of `noreturn`.

`noreturn` A few standard library functions, such as `abort` and `exit`, cannot return. GNU CC knows this automatically. Some programs define their own functions that never return. You

can declare them `noreturn` to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));

void
fatal (...)
{
  ... /* Print error message. */ ...
  exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

Do not assume that registers saved by the calling function are restored before calling the `noreturn` function.

It does not make sense for a `noreturn` function to have a return type other than `void`.

The attribute `noreturn` is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function does not return, which works in the current version and in some older versions, is as follows:

```
typedef void voidfn ();

volatile voidfn fatal;
```

`const` Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `const`. For example,

```
int square (int) __attribute__ ((const));
```

says that the hypothetical function `square` is safe to call fewer times than the program says.

The attribute `const` is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function has no side effects, which works in the current version and in some older versions, is as follows:

```
typedef int intfn ();
extern const intfn square;
```

This approach does not work in GNU C++ from 2.6.0 on, since the language specifies that the '`const`' must be attached to the return value.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to return `void`.

`format (archetype, string-index, first-to-check)`

The `format` attribute specifies that a function takes `printf` or `scanf` style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format,...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter *archetype* determines how the format string is interpreted, and should be either `printf` or `scanf`. The parameter *string-index* specifies which argument is the format string argument (starting from 1), while *first-to-check* is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions which take format strings as arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `vprintf`, `vfprintf` and `vsprintf` whenever such warnings are requested (using '`-Wformat`'), so there is no need to modify the header file '`stdio.h`'.

`section ("section-name")`

Normally, the compiler places the code it generates in the `text` section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__
                  ((section (".init")));
```

puts the function `foobar` in the `.init` section.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`constructor`
`destructor`

The `constructor` attribute causes the function to be called automatically before execution enters `main ()`. Similarly, the `destructor` attribute causes the function to be called automatically after `main ()` has completed or `exit ()` has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

These attributes are not currently implemented for Objective C.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

Some people object to the `__attribute__` feature, suggesting that ANSI C's `#pragma` should be used instead. There are two reasons for not doing this.

1. It is impossible to generate `#pragma` commands from a macro.
2. There is no telling what the same `#pragma` might mean in another compiler.

These two reasons apply to almost any application that might be proposed for `#pragma`. It is basically a mistake to use `#pragma` for *anything*.

6.23 Prototypes and Old-Style Function Definitions

GNU C extends ANSI C to allow a function prototype to override a later old-style non-prototype definition. Consider the following example:

```
/* Use prototypes unless the compiler is old-fashioned.  */
#if __STDC__
#define P(x) x
#else
#define P(x) ()
#endif
```

```
/* Prototype function declaration.   */
int isroot P((uid_t));

/* Old-style function definition.   */
int
isroot (x)   /* ??? lossage here ??? */
     uid_t x;
{
  return x == 0;
}
```

Suppose the type `uid_t` happens to be `short`. ANSI C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition's argument is really an `int`, which does not match the prototype argument type of `short`.

This restriction of ANSI C makes it hard to write code that is portable to traditional C compilers, because the programmer does not know whether the `uid_t` type is `short`, `int`, or `long`. Therefore, in cases like these GNU C allows a prototype to override a later old-style definition. More precisely, in GNU C, a function prototype argument type overrides the argument type specified by a later old-style definition if the former type is the same as the latter type before promotion. Thus in GNU C the above example is equivalent to the following:

```
int isroot (uid_t);

int
isroot (uid_t x)
{
  return x == 0;
}
```

GNU C++ does not support old-style function definitions, so this extension is irrelevant.

6.24 Compiling Functions for Interrupt Calls

When compiling code for certain platforms (currently the Hitachi H8/300 and the Tandem ST-2000), you can instruct GCC that certain functions are meant to be called from hardware interrupts.

To mark a function as callable from interrupt, include the line '`#pragma interrupt`' somewhere before the beginning of the function's definition. (For maximum readability, you might place it immediately before the definition of the appropriate function.) '`#pragma interrupt`' will affect only the next function defined; if you want to define more than one function with this property, include '`#pragma interrupt`' before each of them.

When you define a function with '#pragma interrupt', GCC alters its usual calling convention, to provide the right environment when the function is called from an interrupt. *Such functions cannot be called in the usual way from your program.*

You must use other facilities to actually associate these functions with particular interrupts; GCC can only compile them in the appropriate way.

6.25 Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

On some machines, dollar signs are allowed in identifiers if you specify '-traditional'. On a few systems they are allowed by default, even if you do not use '-traditional'. But they are never allowed if you specify '-ansi'.

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$
lose (test)
```

6.26 The Character ESC in Constants

You can use the sequence '\e' in a string or character constant to stand for the ASCII character ESC.

6.27 Inquiring on Alignment of Types or Variables

The keyword `__alignof__` allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like `sizeof`.

For example, if the target machine requires a `double` value to be aligned on an 8-byte boundary, then `__alignof__ (double)` is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof__ (double)` is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, `__alignof__` reports the *recommended* alignment of a type.

When the operand of `__alignof__` is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof__ (foo1.y)` is probably 2 or 4, the same as `__alignof__ (int)`, even though the data type of `foo1.y` does not itself demand any alignment.

A related feature which lets you specify the alignment of an object is `__attribute__ ((aligned (alignment)))`; see the following section.

6.28 Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Six attributes are currently defined for variables: `aligned`, `mode`, `nocommon`, `packed`, `section`, and `transparent_union`. Other attributes are available for functions (see Section 6.22 “Function Attributes,” page 157) and for types (see Section 6.29 “Type Attributes,” page 166).

You may also specify attributes with ‘__’ preceeding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__` instead of `aligned`.

`aligned (`*alignment*`)`

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On a 68040, this could be used in conjunction with an `asm` expression to access the `move16` instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__
                    ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed. You cannot specify alignment for a typedef name because such a name is just an alias, not a distinct type.

As in the preceeding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`mode (`*`mode`*`)`

This attribute specifies the data type for the declaration—whichever type corresponds to the mode *`mode`*. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of '`byte`' or '`__byte__`' to indicate the mode corresponding to a one-byte integer, '`word`' or '`__word__`' for the mode of a one-word integer, and '`pointer`' or '`__pointer__`' for the mode used to represent pointers.

`nocommon`
: This attribute specifies requests GNU CC not to place a variable “common” but instead to allocate space for it directly. If you specify the '`-fno-common`' flag, GNU CC will do this for all variables.

  Specifying the `nocommon` attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

`packed`
: The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

  Here is a structure in which the field `x` is packed, so that it immediately follows `a`:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

`section ("section-name")`
: Normally, the compiler places the objects it generates in sections like `data` and `bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__
        ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__
        ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__
        ((section ("STACK"))) = { 0 };
int init_data_copy __attribute__
        ((section ("INITDATACOPY"))) = 0;

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data_copy, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

Use the `section` attribute with an *initialized* definition of a *global* variable, as shown in the example. GNU CC issues a warning and otherwise ignores the `section` attribute in uninitialized variable declarations.

You may only use the `section` attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the `common` (or `bss`) section and can be multiply "defined". You can force a variable to be initialized with the '`-fno-common`' flag or the `nocommon` attribute.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`transparent_union`

This attribute, attached to a function argument variable which is a union, means to pass the argument in the same way that the first union member would be passed. You can also use this attribute on a `typedef` for a union data type; then it applies to all function arguments with that type.

To specify multiple attributes, separate them by commas within the double parentheses: for example, '`__attribute__ ((aligned (16), packed))`'.

6.29 Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of `struct` and `union` types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Two attributes are currently defined for types: `aligned`, and `transparent_union`. Other attributes are defined for functions (see Section 6.22 "Function Attributes," page 157) and for variables (see Section 6.28 "Variable Attributes," page 163).

You may also specify any one of these attributes with '`__`' preceeding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__` instead of `aligned`.

The defined type attributes are supported only for `struct` and `union` types. Furthermore, these attributes are only relevant for *complete* struct and union types, and so they are only allowed to appear just past

the closing curly brace of a complete struct or union type *definition*. This point is illustrated by the examples given below.

`aligned (`*alignment*`)`

This attribute specifies a minimum alignment (in bytes) for variables whose type is the relevant struct or union type. For example, the declaration:

```
struct S { short f[3]; } __attribute__
                         ((aligned (8)));
```

forces the compiler to insure (as fas as it can) that each variable whose type is `struct S` will be allocated and aligned *at least* on a 8-byte boundary. On a Sparc, having all variables of type `struct S` aligned to 8-byte boundaries allows the compiler to use the `ldd` and `std` (doubleword load and store) instructions when copying one variable of type `struct S` to another, thus improving run-time efficiency.

Note that the alignment of any given `struct` or `union` type is required by the ANSI C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question. This means that you *can* effectively adjust the alignment of a `struct` or `union` type by attaching an `aligned` attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intutive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.

As in the preceeding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each `short` is 2 bytes, then the size of the entire `struct S` type is 6 bytes. The smallest

power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer artihmetic operations will often be more efficient for efficiently-aligned types than for other types.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`transparent_union`

This attribute, attached to a `union` type definition, indicates that any variable having that union type should, if passed to a function, be passed in the same way that the first union member would be passed. For example:

```
union foo
{
  char a;
  int x[2];
} __attribute__ ((transparent_union));
```

To specify multiple attributes, separate them by commas within the double parentheses: for example, '`__attribute__ ((aligned (16), packed))`'.

6.30 An Inline Function is As Fast As a Macro

By declaring a function `inline`, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation. If you don't use '`-O`', no function is really inline.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write `__inline__` instead of `inline`. See Section 6.35 "Alternate Keywords," page 187.)

You can also make all "simple enough" functions inline with the option '`-finline-functions`'. Note that certain usages in a function definition can make it unsuitable for inline substitution.

Note that in C and Objective C, unlike C++, the `inline` keyword does not affect the linkage of the function.

GNU CC automatically inlines member functions defined within the class body of C++ programs even if they are not explicitly declared `inline`. (You can override this with '`-fno-default-inline`'; see Section 4.5 "Options Controlling C++ Dialect," page 35.)

When a function is both inline and `static`, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option '`-fkeep-inline-functions`'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

GNU C does not inline any functions when not optimizing. It is not clear whether it is better to inline or not, in this case, but we found that a correct implementation when not optimizing was difficult. So we did the easy thing, and turned it off.

6.31 Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has '`"f"`' as its operand constraint, saying that a floating point register is required. The '`=`' in '`=f`' indicates that the operand is an output; all output operands' constraints must use '`=`'. The constraints use the same language used in the machine description (see Section 6.32 "Constraints," page 174).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler

template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions that the compiler itself does not know exist.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm does not support input-output or read-write operands. For this reason, the constraint character '`+`', which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) '`combine`' instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint '`"0"`' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different

register (copying it afterward to foo's own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many assemblers, the register names begin with '%'; to produce one '%' in the assembler code, you must write '%%' in the input.

If your assembler instruction can alter the condition code register, add 'cc' to the list of clobbered registers. GNU CC on some machines represents the condition codes as a specific hardware register; 'cc' serves to name this register. On other machines, the condition code is handled differently, and specifying 'cc' has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add 'memory' to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single asm template, separated either with newlines (written as '\n') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine _foo accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the '&' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '&' for

each output operand that may not overlap an input. See Section 6.32.3 “Modifiers,” page 177.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the asm construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
     : "g" (result)
     : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one asm to another are not supported. The compiler’s optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these asm instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
   __value; })
```

Here the variable __arg is used to make sure that the instruction operates on a proper double value, and to accept only those arguments x which can convert automatically to a double.

Another way to make sure the instruction operates on the correct data type is to use a cast in the asm. This is different from using a variable __arg in that it converts more different types. For example, if the desired type were int, casting the argument to int would accept a pointer with no complaint, while assigning the argument to an int variable named __arg would warn about using a pointer unless the caller explicitly casts it.

If an asm has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren’t used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an asm instruction from being deleted, moved significantly, or combined, by writing the keyword volatile after the asm. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile `asm` instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile `asm` instructions to remain perfectly consecutive. If you want consecutive output, use a single `asm`.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write `__asm__` instead of `asm`. See Section 6.35 "Alternate Keywords," page 187.

6.32 Constraints for `asm` Operands

Here are specific details on what constraint letters you can use with `asm` operands. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

6.32.1 Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

'`m`'
: A memory operand is allowed, with any kind of address that the machine supports in general.

'`o`'
: A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

  For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range

of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter 'o' is valid only when accompanied by both '<' (if the target machine has predecrement addressing) and '>' (if the target machine has preincrement addressing).

'V' A memory operand that is not offsettable. In other words, anything that would fit the 'm' constraint but not the 'o' constraint.

'<' A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed.

'>' A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed.

'r' A register operand is allowed provided that it is in a general register.

'd', 'a', 'f', ...
Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. 'd', 'a' and 'f' are defined on the 68000/68020 to stand for data, address and floating point registers.

'i' An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

'n' An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use 'n' rather than 'i'.

'I', 'J', 'K', ... 'P'
Other letters in the range 'I' through 'P' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, 'I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

'E' An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format

is the same as that of the host machine (on which the compiler is running).

'F' An immediate floating operand (expression code `const_double`) is allowed.

'G', 'H' 'G' and 'H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

's' An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use 's' instead of 'i'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a 'moveq' instruction. We arrange for this to happen by defining the letter 'K' to mean "any integer outside the range -128 to 127", and then specifying 'Ks' in the operand constraints.

'g' Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

'X' Any operand whatsoever is allowed.

'0', '1', '2', . . . '9'

An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles which `asm` distinguishes. For example, an add instruction uses two input operands and an output operand, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

'p' An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

'p' in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

'Q', 'R', 'S', ... 'U'
Letters in the range 'Q' through 'U' may be defined in a machine-dependent fashion to stand for arbitrary operand types.

6.32.2 Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the '?' and '!' characters:

? Disparage slightly the alternative that the '?' appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each '?' that appears in it.

! Disparage severely the alternative that the '!' appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

6.32.3 Constraint Modifier Characters

Here are constraint modifier characters.

'='
: Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

'+'
: Means that this operand is both read and written by the instruction.

  When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. '=' identifies an output; '+' identifies an operand that is both input and output; all other operands are assumed to be input only.

'&'
: Means (in a particular alternative) that this operand is written before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

  '&' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires '&' while others do not. See, for example, the 'movdf' insn of the 68000.

  '&' does not obviate the need to write '='.

'%'
: Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints.

'#'
: Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

6.32.4 Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in `asm` arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures. The most commonly used constraints are 'm' and 'r' (for memory and general-purpose registers respectively; see Section 6.32.1 "Simple Constraints," page 174), and 'I', usually the letter indicating the most common immediate-constant format.

For each machine architecture, the 'config/*machine*.h' file defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for `asm` statements; therefore, some of the constraints are not particularly interesting for `asm`. The constraints are defined through these macros:

`REG_CLASS_FROM_LETTER`
: Register class constraints (usually lower case).

`CONST_OK_FOR_LETTER_P`
: Immediate constant constraints, for non-floating point constants of word size or smaller precision (usually upper case).

`CONST_DOUBLE_OK_FOR_LETTER_P`
: Immediate constant constraints, for all floating point constants and for constants of greater than word size precision (usually upper case).

`EXTRA_CONSTRAINT`
: Special cases of registers or memory. This macro is not required, and is only defined for some machines.

Inspecting these macro definitions in the compiler source for your machine is the best way to be certain you have the right constraints. However, here is a summary of the machine-dependent constraints available on some particular machines.

ARM family—'`arm.h`'

`f` Floating-point register

`F` One of the floating-point constants 0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0 or 10.0

`G` Floating-point constant that would satisfy the constraint '`F`' if it were negated

`I` Integer that is valid as an immediate operand in a data processing instruction. That is, an integer in the range 0 to 255 rotated by a multiple of 2

`J` Integer in the range -4095 to 4095

`K` Integer that satisfies constraint '`I`' when inverted (ones complement)

`L` Integer that satisfies constraint '`I`' when negated (twos complement)

`M` Integer in the range 0 to 32

`Q` A memory reference where the exact address is in a single register ("`m`" is preferable for `asm` statements)

`R` An item in the constant pool

`S` A symbol in the text segment of the current file

AMD 29000 family—'`a29k.h`'

`l`　Local register 0

`b`　Byte Pointer (‘`BP`’) register

`q`　‘`Q`’ register

`h`　Special purpose register

`A`　First accumulator register

`a`　Other accumulator register

`f`　Floating point register

`I`　Constant greater than 0, less than 0x100

`J`　Constant greater than 0, less than 0x10000

`K`　Constant whose high 24 bits are on (1)

`L`　16 bit constant whose high 8 bits are on (1)

`M`　32 bit constant whose high 16 bits are on (1)

`N`　32 bit negative constant that fits in 8 bits

`O`　The constant 0x80000000 or, on the 29050, any 32 bit constant whose low 16 bits are 0.

`P`　16 bit negative constant that fits in 8 bits

`G`
`H`　A floating point constant (in `asm` statements, use the machine independent ‘`E`’ or ‘`F`’ instead)

IBM RS6000—‘`rs6000.h`’

`b`　Address base register

`f`　Floating point register

`h`　‘`MQ`’, ‘`CTR`’, or ‘`LINK`’ register

`q`　‘`MQ`’ register

`c`　‘`CTR`’ register

`l`　‘`LINK`’ register

`x`　‘`CR`’ register (condition register) number 0

`y`　‘`CR`’ register (condition register)

`I`　Signed 16 bit constant

`J`　Constant whose low 16 bits are 0

`K`　Constant whose high 16 bits are 0

`L` Constant suitable as a mask operand

`M` Constant larger than 31

`N` Exact power of 2

`O` Zero

`P` Constant whose negation is a signed 16 bit constant

`G` Floating point constant that can be loaded into a register with one instruction per word

`Q` Memory operand that is an offset from a register (‘`m`’ is preferable for `asm` statements)

Intel 386—‘`i386.h`’

`q` ‘`a`’, `b`, `c`, or `d` register

`A` ‘`a`’, or `d` register (for 64-bit ints)

`f` Floating point register

`t` First (top of stack) floating point register

`u` Second floating point register

`a` ‘`a`’ register

`b` ‘`b`’ register

`c` ‘`c`’ register

`d` ‘`d`’ register

`D` ‘`di`’ register

`S` ‘`si`’ register

`I` Constant in range 0 to 31 (for 32 bit shifts)

`J` Constant in range 0 to 63 (for 64 bit shifts)

`K` ‘`0xff`’

`L` ‘`0xffff`’

`M` 0, 1, 2, or 3 (shifts for `lea` instruction)

`G` Standard 80387 floating point constant

Intel 960—‘`i960.h`’

`f` Floating point register (`fp0` to `fp3`)

`l` Local register (`r0` to `r15`)

`b` Global register (`g0` to `g15`)

`d` Any local or global register

`I` Integers from 0 to 31

`J` 0

`K` Integers from -31 to 0

`G` Floating point 0

`H` Floating point 1

MIPS—'`mips.h`'

`d` General-purpose integer register

`f` Floating-point register (if available)

`h` '`Hi`' register

`l` '`Lo`' register

`x` '`Hi`' or '`Lo`' register

`y` General-purpose integer register

`z` Floating-point status register

`I` Signed 16 bit constant (for arithmetic instructions)

`J` Zero

`K` Zero-extended 16-bit constant (for logic instructions)

`L` Constant with low 16 bits zero (can be loaded with `lui`)

`M` 32 bit constant which requires two instructions to load (a constant which is not '`I`', '`K`', or '`L`')

`N` Negative 16 bit constant

`O` Exact power of two

`P` Positive 16 bit constant

`G` Floating point zero

`Q` Memory reference that can be loaded with more than one instruction ('`m`' is preferable for `asm` statements)

`R` Memory reference that can be loaded with one instruction ('`m`' is preferable for `asm` statements)

`S` Memory reference in external OSF/rose PIC format ('`m`' is preferable for `asm` statements)

Motorola 680x0—'`m68k.h`'

`a` Address register

`d` Data register

`f` 68881 floating-point register, if available

`x` Sun FPA (floating-point) register, if available

`y` First 16 Sun FPA registers, if available

`I` Integer in the range 1 to 8

`J` 16 bit signed number

`K` Signed number whose magnitude is greater than 0x80

`L` Integer in the range -8 to -1

`G` Floating point constant that is not a 68881 constant

`H` Floating point constant that can be used by Sun FPA

SPARC—'`sparc.h`'

`f` Floating-point register

`I` Signed 13 bit constant

`J` Zero

`K` 32 bit constant with the low 12 bits clear (a constant that can be loaded with the `sethi` instruction)

`G` Floating-point zero

`H` Signed 13 bit constant, sign-extended to 32 or 64 bits

`Q` Memory reference that can be loaded with one instruction ('`m`' is more appropriate for `asm` statements)

`S` Constant, or memory address

`T` Memory address aligned to an 8-byte boundary

`U` Even register

6.33 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use `asm` in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
extern func () asm ("FUNC");

func (x, y)
     int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU CC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

6.34 Variables in Specified Registers

GNU C allows you to put a few global variables into specified hardware registers. You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses.

  These local variables are sometimes convenient for use with the extended `asm` feature (see Section 6.31 "Extended Asm," page 170), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the `asm`.)

6.34.1 Defining Global Register Variables

You can define a global register variable in GNU C like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register `a5` would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a "global" register that is not affected magically by the function call mechanism.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e. in a different source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. (If you are prepared to recompile `qsort` with the same global register variable, you can solve this problem.)

If you want to recompile `qsort` or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler

option '`-ffixed-reg`'. You need not actually add a global register declaration to their source code.

A function which can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`. On some machines, however, `longjmp` will not change the value of global register variables. To be portable, the function that called `setjmp` should make other arrangements to save the values of the global register variables, and to restore them in a `longjmp`. This way, the same thing will happen regardless of what `longjmp` does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

On the Sparc, there are reports that g3 ... g7 are suitable registers, but certain library functions, such as `getwd`, as well as the subroutines for division and remainder, modify g3 and g4. g1 and g2 are local temporaries.

On the 68000, a2 ... a5 should be suitable, as should d2 ... d7. Of course, it will not do to use more than a few of those.

6.34.2 Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Naturally the register name is cpu-dependent, but this is not a problem, since specific registers are most often useful with explicit assembler instructions (see Section 6.31 "Extended Asm," page 170). Both of these things generally require that you conditionalize your program according to cpu type.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. However, these registers are made unavailable for use in the reload pass. I would not be surprised if excessive use of this feature leaves the compiler too few available registers to compile certain functions.

6.35 Alternate Keywords

The option '`-traditional`' disables certain keywords; '`-ansi`' disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords `asm`, `typeof` and `inline` cannot be used since they won't work in a program compiled with '`-ansi`', while the keywords `const`, `volatile`, `signed`, `typeof` and `inline` won't work in a program compiled with '`-traditional`'.

The way to solve these problems is to put '__' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, `__const__` instead of `const`, and `__inline__` instead of `inline`.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

'`-pedantic`' causes warnings for many GNU C extensions. You can prevent such warnings within one expression by writing `__extension__` before the expression. `__extension__` has no effect aside from this.

6.36 Incomplete `enum` Types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write

`struct foo` without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.

This extension is not supported by GNU C++.

6.37 Function Names as Strings

GNU CC predefines two string variables to be the name of the current function. The variable `__FUNCTION__` is the name of the function as it appears in the source. The variable `__PRETTY_FUNCTION__` is the name of the function pretty printed in a language specific fashion.

These names are always the same in a C function, but in a C++ function they may be different. For example, this program:

```
extern "C" {
extern int printf (char *, ...);
}

class a {
 public:
  sub (int i)
    {
      printf ("__FUNCTION__ = %s\n", __FUNCTION__);
      printf ("__PRETTY_FUNCTION__ = %s\n", __PRETTY_FUNCTION__);
    }
};

int
main (void)
{
  a ax;
  ax.sub (0);
  return 0;
}
```

gives this output:

```
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = int  a::sub (int)
```

7 Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro `__GNUC__`. You can also use `__GNUG__` to test specifically for GNU C++ (see section "Standard Predefined Macros" in *The C Preprocessor*).

7.1 Named Return Values in C++

GNU C++ extends the function-definition syntax to allow you to specify a name for the result of a function outside the body of the definition, in C++ programs:

```
type
functionname (args) return resultname;
{
  ...
  body
  ...
}
```

You can use this feature to avoid an extra constructor call when a function result has a class type. For example, consider a function `m`, declared as '`X v = m ();`', whose result is of class `X`:

```
X
m ()
{
  X b;
  b.a = 23;
  return b;
}
```

Although `m` appears to have no arguments, in fact it has one implicit argument: the address of the return value. At invocation, the address of enough space to hold `v` is sent in as the implicit argument. Then `b` is constructed and its `a` field is set to the value 23. Finally, a copy constructor (a constructor of the form '`X(X&)`') is applied to `b`, with the (implicit) return value location as the target, so that `v` is now bound to the return value.

But this is wasteful. The local `b` is declared just to hold something that will be copied right out. While a compiler that combined an "elision" algorithm with interprocedural data flow analysis could conceivably eliminate all of this, it is much more practical to allow you to assist the compiler in generating efficient code by manipulating the return

value explicitly, thus avoiding the local variable and copy constructor altogether.

Using the extended GNU C++ function-definition syntax, you can avoid the temporary allocation and copying by naming `r` as your return value at the outset, and assigning to its `a` field directly:

```
X
m () return r;
{
  r.a = 23;
}
```

The declaration of `r` is a standard, proper declaration, whose effects are executed **before** any of the body of `m`.

Functions of this type impose no additional restrictions; in particular, you can execute `return` statements, or return implicitly by reaching the end of the function body ("falling off the edge"). Cases like

```
X
m () return r (23);
{
  return;
}
```

(or even '`X m () return r (23); { }`') are unambiguous, since the return value `r` has been initialized in either case. The following code may be hard to read, but also works predictably:

```
X
m () return r;
{
  X b;
  return b;
}
```

The return value slot denoted by `r` is initialized at the outset, but the statement '`return b;`' overrides this value. The compiler deals with this by destroying `r` (calling the destructor if there is one, or doing nothing if there is not), and then reinitializing `r` with `b`.

This extension is provided primarily to help people who use overloaded operators, where there is a great need to control not just the arguments, but the return values of functions. For classes where the copy constructor incurs a heavy performance penalty (especially in the common case where there is a quick default constructor), this is a major savings. The disadvantage of this extension is that you do not control when the default constructor for the return value is called: it is always called at the beginning.

7.2 Minimum and Maximum Operators in C++

It is very convenient to have operators which return the “minimum” or the “maximum” of two arguments. In GNU C++ (but not in GNU C),

a <? *b*
: is the *minimum*, returning the smaller of the numeric values *a* and *b*;

a >? *b*
: is the *maximum*, returning the larger of the numeric values *a* and *b*.

These operations are not primitive in ordinary C++, since you can use a macro to return the minimum of two things in C++, as in the following example.

```
#define MIN(X,Y) ((X) < (Y) ? : (X) : (Y))
```

You might then use ‘int min = MIN (i, j);’ to set *min* to the minimum value of variables *i* and *j*.

However, side effects in X or Y may cause unintended behavior. For example, MIN (i++, j++) will fail, incrementing the smaller counter twice. A GNU C extension allows you to write safe macros that avoid this kind of problem (see Section 6.6 “Naming an Expression’s Type,” page 146). However, writing MIN and MAX as macros also forces you to use function-call notation notation for a fundamental arithmetic operation. Using GNU C++ extensions, you can write ‘int min = i <? j;’ instead.

Since <? and >? are built into the compiler, they properly handle expressions with side-effects; ‘int min = i++ <? j++;’ works correctly.

7.3 goto and Destructors in GNU C++

In C++ programs, you can safely use the goto statement. When you use it to exit a block which contains aggregates requiring destructors, the destructors will run before the goto transfers control. (In ANSI C++, goto is restricted to targets within the current block.)

The compiler still forbids using goto to *enter* a scope that requires constructors.

7.4 Declarations and Definitions in One Header

C++ object definitions can be quite complex. In principle, your source code will need two kinds of things for each object that you use across more than one source file. First, you need an *interface* specification, describing its structure with type declarations and function prototypes. Second, you need the *implementation* itself. It can be tedious to maintain a separate interface description in a header file, in parallel to the

actual implementation. It is also dangerous, since separate interface and implementation definitions may not remain parallel.

With GNU C++, you can use a single header file for both purposes.

> *Warning:* The mechanism to specify this is in transition. For the nonce, you must use one of two `#pragma` commands; in a future release of GNU C++, an alternative mechanism will make these `#pragma` commands unnecessary.

The header file contains the full definitions, but is marked with '`#pragma interface`' in the source code. This allows the compiler to use the header file only as an interface specification when ordinary source files incorporate it with `#include`. In the single source file where the full implementation belongs, you can use either a naming convention or '`#pragma implementation`' to indicate this alternate use of the header file.

`#pragma interface`
`#pragma interface "`*`subdir`*`/`*`objects`*`.h"`

Use this directive in *header files* that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information, and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing '`#pragma interface`' is included in a compilation, this auxiliary information will not be generated (unless the main input source file itself uses '`#pragma implementation`'). Instead, the object files will contain references to be resolved at link time.

The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to '`#pragma implementation`'.

`#pragma implementation`
`#pragma implementation "`*`objects`*`.h"`

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use '`#pragma interface`'. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use '#pragma implementation' with no argument, it applies to an include file with the same basename[1] as your source file. For example, in 'allclass.cc', '#pragma implementation' by itself is equivalent to '#pragma implementation "allclass.h"'.

In versions of GNU C++ prior to 2.6.0 'allclass.h' was treated as an implementation file whenever you would include it from 'allclass.cc' even if you never specified '#pragma implementation'. This was deemed to be more trouble than it was worth, however, and disabled.

If you use an explicit '#pragma implementation', it must appear in your source file *before* you include the affected header files.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use '#include' to include the header file; '#pragma implementation' only specifies how to use the file—it doesn't actually include it.)

There is no way to split up the contents of a single header file into multiple implementation files.

'#pragma implementation' and '#pragma interface' also have an effect on function inlining.

If you define a class in a header file marked with '#pragma interface', the effect on a function defined in that class is similar to an explicit extern declaration—the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as '#pragma implementation', the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with '-fno-implement-inlines'. If any calls were not inlined, you will get linker errors.

7.5 Where's the Template?

C++ templates are the first language feature to require more intelligence from the environment than one usually finds on a UNIX system.

[1] A file's *basename* was the name stripped of all leading path information and of trailing suffixes, such as '.h' or '.C' or '.cc'.

Somehow the compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which I will refer to as the Borland model and the Cfront model.

Borland model
: Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; template instances are emitted in each translation unit that uses them, and they are collapsed together at run time. The advantage of this model is that the linker only has to consider the object files themselves; there is no external complexity to worry about. This disadvantage is that compilation time is increased because the template code is being compiled repeatedly. Code written for this model tends to include definitions of all member templates in the header file, since they must be seen to be compiled.

Cfront model
: The AT&T C++ translator, Cfront, solved the template instantiation problem by creating the notion of a template repository, an automatically maintained place where template instances are stored. As individual object files are built, notes are placed in the repository to record where templates and potential type arguments were seen so that the subsequent instantiation step knows where to find them. At link time, any needed instances are generated and linked in. The advantages of this model are more optimal compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are vastly increased complexity, and thus potential for error; theoretically, this should be just as transparent, but in practice it has been very difficult to build multiple programs in one directory and one program in multiple directories using Cfront. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which is magically found by the link preprocessor when a template needs to be instantiated.

Currently, g++ implements neither automatic model. The g++ team hopes to have a repository working for 2.7.0. In the mean time, you have three options for dealing with template instantiations:

1. Do nothing. Pretend g++ does implement automatic instantiation management. Code written for the Borland model will work fine, but each translation unit will contain instances of each of the templates

it uses. In a large program, this can lead to an unacceptable amount of code duplication.

2. Add '#pragma interface' to all files containing template definitions. For each of these files, add '#pragma implementation "*filename*"' to the top of some '.C' file which '#include's it. Then compile everything with -fexternal-templates. The templates will then only be expanded in the translation unit which implements them (i.e. has a '#pragma implementation' line for the file where they live); all other files will use external references. If you're lucky, everything should work properly. If you get undefined symbol errors, you need to make sure that each template instance which is used in the program is used in the file which implements that template. If you don't have any use for a particular instance in that file, you can just instantiate it explicitly, using the syntax from the latest C++ working paper:

   ```
   template class A<int>;
   template ostream& operator << (ostream&, const A<int>&);
   ```

   This strategy will work with code written for either model. If you are using code written for the Cfront model, the file containing a class template and the file containing its member templates should be implemented in the same translation unit.

   A slight variation on this approach is to use the flag -falt-external-templates instead; this flag causes template instances to be emitted in the translation unit that implements the header where they are first instantiated, rather than the one which implements the file where the templates are defined. This header must be the same in all translation units, or things are likely to break.

   See Section 7.4 "Declarations and Definitions in One Header," page 191, for more discussion of these pragmas.

3. Explicitly instantiate all the template instances you use, and compile with -fno-implicit-templates. This is probably your best bet; it may require more knowledge of exactly which templates you are using, but it's less mysterious than the previous approach, and it doesn't require any '#pragma's or other g++-specific code. You can scatter the instantiations throughout your program, you can create one big file to do all the instantiations, or you can create tiny files like

   ```
   #include "Foo.h"
   #include "Foo.cc"

   template class Foo<int>;
   ```

   for each instance you need, and create a template instantiation library from those. I'm partial to the last, but your mileage may vary. If you are using Cfront-model code, you can probably get away with

not using -fno-implicit-templates when compiling files that don't '`#include`' the member template definitions.

7.6 Type Abstraction using Signatures

In GNU C++, you can use the keyword `signature` to define a completely abstract class interface as a datatype. You can connect this abstraction with actual classes using signature pointers. If you want to use signatures, run the GNU compiler with the '`-fhandle-signatures`' command-line option. (With this option, the compiler reserves a second keyword `sigof` as well, for a future extension.)

Roughly, signatures are type abstractions or interfaces of classes. Some other languages have similar facilities. C++ signatures are related to ML's signatures, Haskell's type classes, definition modules in Modula-2, interface modules in Modula-3, abstract types in Emerald, type modules in Trellis/Owl, categories in Scratchpad II, and types in POOL-I. For a more detailed discussion of signatures, see *Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism* by Gerald Baumgartner and Vincent F. Russo (Tech report CSD–TR–93–059, Dept. of Computer Sciences, Purdue University, September 1993, to appear in *Software Practice & Experience*). You can get the tech report by anonymous FTP from `ftp.cs.purdue.edu` in '`pub/reports/TR93-059.PS.Z`'.

Syntactically, a signature declaration is a collection of member function declarations and nested type declarations. For example, this signature declaration defines a new abstract type `S` with member functions '`int foo ()`' and '`int bar (int)`':

```
signature S
{
  int foo ();
  int bar (int);
};
```

Since signature types do not include implementation definitions, you cannot write an instance of a signature directly. Instead, you can define a pointer to any class that contains the required interfaces as a *signature pointer*. Such a class *implements* the signature type.

To use a class as an implementation of `S`, you must ensure that the class has public member functions '`int foo ()`' and '`int bar (int)`'. The class can have other member functions as well, public or not; as long as it offers what's declared in the signature, it is suitable as an implementation of that signature type.

For example, suppose that `C` is a class that meets the requirements of signature `S` (`C` *conforms to* `S`). Then

```
C obj;
S * p = &obj;
```

defines a signature pointer `p` and initializes it to point to an object of type `C`. The member function call '`int i = p->foo ();`' executes '`obj.foo ()`'.

Abstract virtual classes provide somewhat similar facilities in standard C++. There are two main advantages to using signatures instead:

1. Subtyping becomes independent from inheritance. A class or signature type `T` is a subtype of a signature type `S` independent of any inheritance hierarchy as long as all the member functions declared in `S` are also found in `T`. So you can define a subtype hierarchy that is completely independent from any inheritance (implementation) hierarchy, instead of being forced to use types that mirror the class inheritance hierarchy.
2. Signatures allow you to work with existing class hierarchies as implementations of a signature type. If those class hierarchies are only available in compiled form, you're out of luck with abstract virtual classes, since an abstract virtual class cannot be retrofitted on top of existing class hierarchies. So you would be required to write interface classes as subtypes of the abstract virtual class.

There is one more detail about signatures. A signature declaration can contain member function *definitions* as well as member function declarations. A signature member function with a full definition is called a *default implementation*; classes need not contain that particular interface in order to conform. For example, a class `C` can conform to the signature

```
signature T
{
  int f (int);
  int f0 () { return f (0); };
};
```

whether or not `C` implements the member function '`int f0 ()`'. If you define `C::f0`, that definition takes precedence; otherwise, the default implementation `S::f0` applies.

8 gcov: a Test Coverage Program

gcov is a tool you can use, together with GNU CC, to test code coverage in your programs. gcov is free software, but for the moment it is only available from Cygnus Support (pending discussions with the FSF about how they think Cygnus should *really* write it).

This chapter describes version 1.5 of gcov.

Jim Wilson wrote gcov, and the original form of this note. Pat McGregor edited the documentation.

8.1 Introduction to gcov

gcov is a test coverage program. Use it in concert with GNU CC to analyze your programs to help create more efficient, faster running code. You can use gcov as a profiling tool, to help discover where your optimization efforts will best affect your code. You can also use gcov in concert with the other profiling tool, gprof, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help you analyze your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. gcov helps you determine where to work on optimization.

Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

You should compile your code without optimization if you plan to use gcov, because the optimization, by combining some lines of code into one function, may not give you as much information as you need to look for 'hot spots' where the code is using a great deal of computer time. Likewise, because gcov accumulates statistics by line (at the lowest resolution), it works best with a programming style that places only one statement on each line. If you use complicated macros that expand to loops or to other control structures, the statistics are less helpful—they

only report on the line where the macro call appears. If your complex macros behave like functions, you can replace them with inline functions to solve this problem.

`gcov` creates a logfile called '*sourcename*.gcov' which indicates how many times each line of a source file '*sourcename*.c' has executed. You can use these logfiles in conjuction with `gprof` to aid in fine-tuning the performance of your programs. `gprof` gives timing information you can use along with the information you get from `gcov`.

`gcov` works only on code compiled with GNU CC; it is not compatible with any other profiling or test coverage mechanism.

8.2 Invoking gcov

```
gcov [-b] [-v] [-n] [-l] [-f] [-o directory] sourcefile
```

`-b`
: Write branch frequencies to the output file. Write branch summary info to standard output. This option allows you to see how often each branch was taken.

`-v`
: Display the `gcov` version number (on the standard error stream).

`-n`
: Do not create the `gcov` output file.

`-l`
: Create long file names for included source files. For example, if the header file 'x.h' contains code, and was included in the file 'a.c', then running `gcov` on the file 'a.c' will produce an output file called 'a.c.x.h.gcov' instead of 'x.h.gcov'. This can be useful if 'x.h' is included in multiple source files.

`-f`
: Output summaries for each function in addition to the file level summary.

`-o`
: The directory where the object files live. Gcov will search for .bb, .bbg, and .da files in this directory.

To use gcov, first compile your program with two special GNU CC options: '-fprofile-arcs -ftest-coverage'. Then run the program. Then run gcov with your program's source file names as arguments. For example, if your program is called 'tmp.c', this is what you see when you use the basic gcov facility:

```
$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
$ gcov tmp.c
 87.50% of 8 source lines executed in file tmp.c
Creating tmp.c.gcov.
```

The file 'tmp.c.gcov' contains output from gcov. Here is a sample:

```
                main()
                {
           1      int i, total;

           1      total = 0;

          11      for (i = 0; i < 10; i++)
          10        total += i;

           1      if (total != 45)
      ######        printf ("Failure\n");
                  else
           1        printf ("Success\n");
           1    }
```

When you use the '-b' option, your output looks like this:

```
$ gcov -b tmp.c
 87.50% of 8 source lines executed in file tmp.c
 80.00% of 5 branches executed in file tmp.c
 80.00% of 5 branches taken at least once in file tmp.c
 50.00% of 2 calls executed in file tmp.c
Creating tmp.c.gcov.
```

Here is a sample of a resulting 'tmp.c.gcov' file:

```
                main()
                {
           1      int i, total;

           1      total = 0;

          11      for (i = 0; i < 10; i++)
branch 0 taken = 91%
branch 1 taken = 100%
branch 2 taken = 100%
          10        total += i;

           1      if (total != 45)
branch 0 taken = 100%
      ######        printf ("Failure\n");
call 0 never executed
```

```
branch 1 never executed
                  else
          1         printf ("Success\n");
call 0 returns = 100%
          1     }
```

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single source line if there are multiple basic blocks that end on that line. In this case, the branches and calls are each given a number. There is no simple way to map these branches and calls back to source constructs. In general, though, the lowest numbered branch or call will correspond to the leftmost construct on the source line.

For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was taken divided by the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed.

For a call, if it was executed at least once, then a percentage indicating the number of times the call returned divided by the number of times the call was executed will be printed. This will usually be 100%, but may be less for functions call `exit` or `longjmp`, and thus may not return everytime they are called.

8.3 Using gcov with GCC Optimization

If you plan to use `gcov` to help optimize your code, you must first compile your program with two special GNU CC options: '`-fprofile-arcs -ftest-coverage`'. Aside from that, you can use any other GNU CC options; but if you want to prove that every single line in your program was executed, you should not compile with optimization at the same time. On some machines the optimizer can eliminate some simple code lines by combining them with other lines. For example, code like this:

```
if (a != b)
  c = 1;
else
  c = 0;
```

can be compiled into one instruction on some machines. In this case, there is no way for `gcov` to calculate separate execution counts for each line because there isn't separate code for each line. Hence the `gcov` output looks like this if you compiled the program with optimization:

```
      100  if (a != b)
      100    c = 1;
      100  else
      100    c = 0;
```

The output shows that this block of code, combined by optimization, executed 100 times. In one sense this result is correct, because there was only one instruction representing all four of these lines. However, the output does not indicate how many times the result was 0 and how many times the result was 1.

9 Known Causes of Trouble with GNU CC

This section describes known problems that affect users of GNU CC. Most of these are not GNU CC bugs per se—if they were, we would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

9.1 Actual Bugs We Haven't Fixed Yet

- The `fixincludes` script interacts badly with automounters; if the directory of system header files is automounted, it tends to be unmounted while `fixincludes` is running. This would seem to be a bug in the automounter. We don't know any good way to work around it.
- The `fixproto` script will sometimes add prototypes for the `sigsetjmp` and `siglongjmp` functions that reference the `jmp_buf` type before that type is defined. To work around this, edit the offending file and place the typedef in front of the prototypes.
- There are several obscure case of mis-using struct, union, and enum tags that are not detected as errors by the compiler.
- When '`-pedantic-errors`' is specified, GNU C will incorrectly give an error message when a function name is specified in an expression involving the comma operator.
- Loop unrolling doesn't work properly for certain C++ programs. This is a bug in the C++ front end. It sometimes emits incorrect debug info, and the loop unrolling code is unable to recover from this error.

9.2 Installation Problems

This is a list of problems (and some apparent problems which don't really mean anything is wrong) that show up during installation of GNU CC.

- On certain systems, defining certain environment variables such as `CC` can interfere with the functioning of `make`.
- If you encounter seemingly strange errors when trying to build the compiler in a directory other than the source directory, it could be because you have previously configured the compiler in the source directory. Make sure you have done all the necessary preparations. See Section 5.2 "Other Dir," page 127.

- If you build GNU CC on a BSD system using a directory stored in a System V file system, problems may occur in running `fixincludes` if the System V file system doesn't support symbolic links. These problems result in a failure to fix the declaration of `size_t` in 'sys/types.h'. If you find that `size_t` is a signed type and that type mismatches occur, this could be the cause.

  The solution is not to use such a directory for building GNU CC.

- In previous versions of GNU CC, the `gcc` driver program looked for `as` and `ld` in various places; for example, in files beginning with '/usr/local/lib/gcc-'. GNU CC version 2 looks for them in the directory '/usr/local/lib/gcc-lib/*target*/*version*'.

  Thus, to use a version of `as` or `ld` that is not the system default, for example `gas` or GNU `ld`, you must put them in that directory (or make links to them from that directory).

- Some commands executed when making the compiler may fail (return a non-zero status) and be ignored by `make`. These failures, which are often due to files that were not found, are expected, and can safely be ignored.

- It is normal to have warnings in compiling certain files about unreachable code and about enumeration type clashes. These files' names begin with 'insn-'. Also, 'real.c' may get some warnings that you can ignore.

- Sometimes `make` recompiles parts of the compiler when installing the compiler. In one case, this was traced down to a bug in `make`. Either ignore the problem or switch to GNU Make.

- If you have installed a program known as purify, you may find that it causes errors while linking `enquire`, which is part of building GNU CC. The fix is to get rid of the file `real-ld` which purify installs—so that GNU CC won't try to use it.

- On Linux SLS 1.01, there is a problem with 'libc.a': it does not contain the obstack functions. However, GNU CC assumes that the obstack functions are in 'libc.a' when it is the GNU C library. To work around this problem, change the `__GNU_LIBRARY__` conditional around line 31 to '#if 1'.

- On some 386 systems, building the compiler never finishes because `enquire` hangs due to a hardware problem in the motherboard—it reports floating point exceptions to the kernel incorrectly. You can install GNU CC except for 'float.h' by patching out the command to run `enquire`. You may also be able to fix the problem for real by getting a replacement motherboard. This problem was observed in Revision E of the Micronics motherboard, and is fixed in Revision F. It has also been observed in the MYLEX MXA-33 motherboard.

If you encounter this problem, you may also want to consider removing the FPU from the socket during the compilation. Alternatively, if you are running SCO Unix, you can reboot and force the FPU to be ignored. To do this, type 'hd(40)unix auto ignorefpu'.

- On some 386 systems, GNU CC crashes trying to compile 'enquire.c'. This happens on machines that don't have a 387 FPU chip. On 386 machines, the system kernel is supposed to emulate the 387 when you don't have one. The crash is due to a bug in the emulator.

  One of these systems is the Unix from Interactive Systems: 386/ix. On this system, an alternate emulator is provided, and it does work. To use it, execute this command as super-user:

  ```
  ln /etc/emulator.rel1 /etc/emulator
  ```

  and then reboot the system. (The default emulator file remains present under the name 'emulator.dflt'.)

  Try using '/etc/emulator.att', if you have such a problem on the SCO system.

  Another system which has this problem is Esix. We don't know whether it has an alternate emulator that works.

  On NetBSD 0.8, a similar problem manifests itself as these error messages:

  ```
  enquire.c: In function 'fprop':
  enquire.c:2328: floating overflow
  ```

- On SCO systems, when compiling GNU CC with the system's compiler, do not use '-O'. Some versions of the system's compiler miscompile GNU CC with '-O'.
- Sometimes on a Sun 4 you may observe a crash in the program genflags or genoutput while building GNU CC. This is said to be due to a bug in sh. You can probably get around it by running genflags or genoutput manually and then retrying the make.
- On Solaris 2, executables of GNU CC version 2.0.2 are commonly available, but they have a bug that shows up when compiling current versions of GNU CC: undefined symbol errors occur during assembly if you use '-g'.

  The solution is to compile the current version of GNU CC without '-g'. That makes a working compiler which you can use to recompile with '-g'.
- Solaris 2 comes with a number of optional OS packages. Some of these packages are needed to use GNU CC fully. If you did not install all optional packages when installing Solaris, you will need to verify that the packages that GNU CC needs are installed.

To check whether an optional package is installed, use the `pkginfo` command. To add an optional package, use the `pkgadd` command. For further details, see the Solaris documentation.

For Solaris 2.0 and 2.1, GNU CC needs six packages: '`SUNWarc`', '`SUNWbtool`', '`SUNWesu`', '`SUNWhea`', '`SUNWlibm`', and '`SUNWtoo`'.

For Solaris 2.2, GNU CC needs an additional seventh package: '`SUNWsprot`'.

- On Solaris 2, trying to use the linker and other tools in '`/usr/ucb`' to install GNU CC has been observed to cause trouble. For example, the linker may hang indefinitely. The fix is to remove '`/usr/ucb`' from your `PATH`.
- If you use the 1.31 version of the MIPS assembler (such as was shipped with Ultrix 3.1), you will need to use the -fno-delayed-branch switch when optimizing floating point code. Otherwise, the assembler will complain when the GCC compiler fills a branch delay slot with a floating point instruction, such as `add.d`.
- If on a MIPS system you get an error message saying "does not have gp sections for all it's [sic] sectons [sic]", don't worry about it. This happens whenever you use GAS with the MIPS linker, but there is not really anything wrong, and it is okay to use the output file. You can stop such warnings by installing the GNU linker.

  It would be nice to extend GAS to produce the gp tables, but they are optional, and there should not be a warning about their absence.
- In Ultrix 4.0 on the MIPS machine, '`stdio.h`' does not work with GNU CC at all unless it has been fixed with `fixincludes`. This causes problems in building GNU CC. Once GNU CC is installed, the problems go away.

  To work around this problem, when making the stage 1 compiler, specify this option to Make:

  ```
  GCC_FOR_TARGET="./xgcc -B./ -I./include"
  ```

  When making stage 2 and stage 3, specify this option:

  ```
  CFLAGS="-g -I./include"
  ```
- Users have reported some problems with version 2.0 of the MIPS compiler tools that were shipped with Ultrix 4.1. Version 2.10 which came with Ultrix 4.2 seems to work fine.

  Users have also reported some problems with version 2.20 of the MIPS compiler tools that were shipped with RISC/os 4.x. The earlier version 2.11 seems to work fine.
- Some versions of the MIPS linker will issue an assertion failure when linking code that uses `alloca` against shared libraries on RISC-OS 5.0, and DEC's OSF/1 systems. This is a bug in the linker, that is supposed to be fixed in future revisions. To protect against

this, GNU CC passes '-non_shared' to the linker unless you pass an explicit '-shared' or '-call_shared' switch.

- On System V release 3, you may get this error message while linking:

```
ld fatal: failed to write symbol name something
 in strings table for file whatever
```

  This probably indicates that the disk is full or your ULIMIT won't allow the file to be as large as it needs to be.

  This problem can also result because the kernel parameter MAXUMEM is too small. If so, you must regenerate the kernel and make the value much larger. The default value is reported to be 1024; a value of 32768 is said to work. Smaller values may also work.

- On System V, if you get an error like this,

```
/usr/local/lib/bison.simple: In function 'yyparse':
/usr/local/lib/bison.simple:625: virtual memory exhausted
```

  that too indicates a problem with disk space, ULIMIT, or MAXUMEM.

- Current GNU CC versions probably do not work on version 2 of the NeXT operating system.
- On NeXTStep 3.0, the Objective C compiler does not work, due, apparently, to a kernel bug that it happens to trigger. This problem does not happen on 3.1.
- On the Tower models 4*n*0 and 6*n*0, by default a process is not allowed to have more than one megabyte of memory. GNU CC cannot compile itself (or many other programs) with '-O' in that much memory.

  To solve this problem, reconfigure the kernel adding the following line to the configuration file:

```
MAXUMEM = 4096
```

- On HP 9000 series 300 or 400 running HP-UX release 8.0, there is a bug in the assembler that must be fixed before GNU CC can be built. This bug manifests itself during the first stage of compilation, while building 'libgcc2.a':

```
_floatdisf
cc1: warning: '-g' option
            not supported on this version of GCC
cc1: warning: '-g1' option
            not supported on this version of GCC
./xgcc: Internal compiler error:
            program as got fatal signal 11
```

  'archive/cph/hpux-8.0-assembler', a patched version of the assembler, is available by anonymous ftp from altdorf.ai.mit.edu. If you have HP software support, the patch can also be obtained directly from HP, as described in the following note:

  > This is the patched assembler, to patch SR#1653-010439, where the assembler aborts on floating point constants.

> The bug is not really in the assembler, but in the shared library version of the function “cvtnum(3c)”. The bug on “cvtnum(3c)” is SR#4701-078451. Anyway, the attached assembler uses the archive library version of “cvtnum(3c)” and thus does not exhibit the bug.

This patch is also known as PHCO_4484.

- On HP-UX version 8.05, but not on 8.07 or more recent versions, the `fixproto` shell script triggers a bug in the system shell. If you encounter this problem, upgrade your operating system or use BASH (the GNU shell) to run `fixproto`.
- Some versions of the Pyramid C compiler are reported to be unable to compile GNU CC. You must use an older version of GNU CC for bootstrapping. One indication of this problem is if you get a crash when GNU CC compiles the function `muldi3` in file ‘`libgcc2.c`’.

  You may be able to succeed by getting GNU CC version 1, installing it, and using it to compile GNU CC version 2. The bug in the Pyramid C compiler does not seem to affect GNU CC version 1.
- There may be similar problems on System V Release 3.1 on 386 systems.
- On the Intel Paragon (an i860 machine), if you are using operating system version 1.0, you will get warnings or errors about redefinition of `va_arg` when you build GNU CC.

  If this happens, then you need to link most programs with the library ‘`iclib.a`’. You must also modify ‘`stdio.h`’ as follows: before the lines

  ```
  #if     defined(__i860__) && !defined(_VA_LIST)
  #include <va_list.h>
  ```

  insert the line

  ```
  #if __PGC__
  ```

  and after the lines

  ```
  extern int  vprintf(const char *, va_list );
  extern int  vsprintf(char *, const char *, va_list );
  #endif
  ```

  insert the line

  ```
  #endif /* __PGC__ */
  ```

  These problems don’t exist in operating system version 1.1.
- On the Altos 3068, programs compiled with GNU CC won’t work unless you fix a kernel bug. This happens using system versions V.2.2 1.0gT1 and V.2.2 1.0e and perhaps later versions as well. See the file ‘`README.ALTOS`’.
- You will get several sorts of compilation and linking errors on the we32k if you don’t follow the special instructions. See Section 5.1 “Configurations,” page 111.

- A bug in the HP-UX 8.05 (and earlier) shell will cause the fixproto program to report an error of the form:

  ```
  ./fixproto: sh internal 1K buffer overflow
  ```

  To fix this, change the first line of the fixproto script to look like:

  ```
  #!/bin/ksh
  ```

9.3 Cross-Compiler Problems

You may run into problems with cross compilation on certain machines, for several reasons.

- Cross compilation can run into trouble for certain machines because some target machines' assemblers require floating point numbers to be written as *integer* constants in certain contexts.

  The compiler writes these integer constants by examining the floating point value as an integer and printing that integer, because this is simple to write and independent of the details of the floating point representation. But this does not work if the compiler is running on a different machine with an incompatible floating point format, or even a different byte-ordering.

  In addition, correct constant folding of floating point values requires representing them in the target machine's format. (The C standard does not quite require this, but in practice it is the only way to win.)

  It is now possible to overcome these problems by defining macros such as `REAL_VALUE_TYPE`. But doing so is a substantial amount of work for each target machine. See section "Cross Compilation and Floating Point Format" in *Using and Porting GCC*.
- At present, the program '`mips-tfile`' which adds debug support to object files on MIPS systems does not work in a cross compile environment.

9.4 Interoperation

This section lists various difficulties encountered in using GNU C or GNU C++ together with other compilers or with the assemblers, linkers, libraries and debuggers on certain systems.

- Objective C does not work on the RS/6000.
- GNU C++ does not do name mangling in the same way as other C++ compilers. This means that object files compiled with one compiler cannot be used with another.

  This effect is intentional, to protect you from more subtle problems. Compilers differ as to many internal details of C++ implementation,

including: how class instances are laid out, how multiple inheritance is implemented, and how virtual function calls are handled. If the name encoding were made the same, your programs would link against libraries provided from other compilers—but the programs would then crash when run. Incompatible libraries are then detected at link time, rather than at run time.

- Older GDB versions sometimes fail to read the output of GNU CC version 2. If you have trouble, get GDB version 4.4 or later.
- DBX rejects some files produced by GNU CC, though it accepts similar constructs in output from PCC. Until someone can supply a coherent description of what is valid DBX input and what is not, there is nothing I can do about these problems. You are on your own.
- The GNU assembler (GAS) does not support PIC. To generate PIC code, you must use some other assembler, such as '/bin/as'.
- On some BSD systems, including some versions of Ultrix, use of profiling causes static variable destructors (currently used only in C++) not to be run.
- Use of '-I/usr/include' may cause trouble.

  Many systems come with header files that won't work with GNU CC unless corrected by fixincludes. The corrected header files go in a new directory; GNU CC searches this directory before '/usr/include'. If you use '-I/usr/include', this tells GNU CC to search '/usr/include' earlier on, before the corrected headers. The result is that you get the uncorrected header files.

  Instead, you should use these options (when compiling C programs):

  ```
  -I/usr/local/lib/gcc-lib/target/version/include   /
              -I/usr/include
  ```

  For C++ programs, GNU CC also uses a special directory that defines C++ interfaces to standard C subroutines. This directory is meant to be searched *before* other standard include directories, so that it takes precedence. If you are compiling C++ programs and specifying include directories explicitly, use this option first, then the two options above:

  ```
  -I/usr/local/lib/g++-include
  ```

- On some SGI systems, when you use '-lgl_s' as an option, it gets translated magically to '-lgl_s -lX11_s -lc_s'. Naturally, this does not happen when you use GNU CC. You must specify all three options explicitly.
- On a Sparc, GNU CC aligns all values of type double on an 8-byte boundary, and it expects every double to be so aligned. The Sun compiler usually gives double values 8-byte alignment, with one exception: function arguments of type double may not be aligned.

As a result, if a function compiled with Sun CC takes the address of an argument of type `double` and passes this pointer of type `double *` to a function compiled with GNU CC, dereferencing the pointer may cause a fatal signal.

One way to solve this problem is to compile your entire program with GNU CC. Another solution is to modify the function that is compiled with Sun CC to copy the argument into a local variable; local variables are always properly aligned. A third solution is to modify the function that uses the pointer to dereference it via the following function `access_double` instead of directly with '`*`':

```
inline double
access_double (double *unaligned_ptr)
{
  union d2i { double d; int i[2]; };

  union d2i *p = (union d2i *) unaligned_ptr;
  union d2i u;

  u.i[0] = p->i[0];
  u.i[1] = p->i[1];

  return u.d;
}
```

Storing into the pointer can be done likewise with the same union.

- On Solaris, the `malloc` function in the '`libmalloc.a`' library may allocate memory that is only 4 byte aligned. Since GNU CC on the Sparc assumes that doubles are 8 byte aligned, this may result in a fatal signal if doubles are stored in memory allocated by the '`libmalloc.a`' library.

  The solution is to not use the '`libmalloc.a`' library. Use instead `malloc` and related functions from '`libc.a`'; they do not have this problem.
- Sun forgot to include a static version of '`libdl.a`' with some versions of SunOS (mainly 4.1). This results in undefined symbols when linking static binaries (that is, if you use '`-static`'). If you see undefined symbols `_dlclose`, `_dlsym` or `_dlopen` when linking, compile and link against the file '`mit/util/misc/dlsym.c`' from the MIT version of X windows.
- The 128-bit long double format that the Sparc port supports currently works by using the architecturally defined quad-word floating point instructions. Since there is no hardware that supports these instructions they must be emulated by the operating system. Long doubles do not work in Sun OS versions 4.0.3 and earlier, because the kernel eumulator uses an obsolete and incompatible format. Long doubles do not work in Sun OS versions 4.1.1 to 4.1.3 because of

emululator bugs that cause random unpredicatable failures. Long doubles appear to work in Sun OS 5.x (Solaris 2.x).

- On HP-UX version 9.01 on the HP PA, the HP compiler `cc` does not compile GNU CC correctly. We do not yet know why. However, GNU CC compiled on earlier HP-UX versions works properly on HP-UX 9.01 and can compile itself properly on 9.01.
- On the HP PA machine, ADB sometimes fails to work on functions compiled with GNU CC. Specifically, it fails to work on functions that use `alloca` or variable-size arrays. This is because GNU CC doesn't generate HP-UX unwind descriptors for such functions. It may even be impossible to generate them.
- Debugging ('`-g`') is not supported on the HP PA machine, unless you use the preliminary GNU tools (see Chapter 5 "Installation," page 103).
- Taking the address of a label may generate errors from the HP-UX PA assembler. GAS for the PA does not have this problem.
- Using floating point parameters for indirect calls to static functions will not work when using the HP assembler. There simply is no way for GCC to specify what registers hold arguments for static functions when using the HP assembler. GAS for the PA does not have this problem.
- For some very large functions you may receive errors from the HP linker complaining about an out of bounds unconditional branch offset. Fixing this problem correctly requires fixing problems in GNU CC and GAS. We hope to fix this in time for GNU CC 2.6. Until then you can work around by making your function smaller, and if you are using GAS, splitting the function into multiple source files may be necessary.
- GNU CC compiled code sometimes emits warnings from the HP-UX assembler of the form:

  ```
  (warning) Use of GR3 when
    frame >= 8192 may cause conflict.
  ```

  These warnings are harmless and can be safely ignored.
- The current version of the assembler ('`/bin/as`') for the RS/6000 has certain problems that prevent the '`-g`' option in GCC from working. Note that '`Makefile.in`' uses '`-g`' by default when compiling '`libgcc2.c`'.

  IBM has produced a fixed version of the assembler. The upgraded assembler unfortunately was not included in any of the AIX 3.2 update PTF releases (3.2.2, 3.2.3, or 3.2.3e). Users of AIX 3.1 should request PTF U403044 from IBM and users of AIX 3.2 should request PTF U416277. See the file '`README.RS6000`' for more details on these updates.

You can test for the presense of a fixed assembler by using the command

```
as -u < /dev/null
```

If the command exits normally, the assembler fix already is installed. If the assembler complains that "-u" is an unknown flag, you need to order the fix.

- On the IBM RS/6000, compiling code of the form

```
extern int foo;

... foo ...

static int foo;
```

will cause the linker to report an undefined symbol foo. Although this behavior differs from most other systems, it is not a bug because redefining an extern variable as static is undefined in ANSI C.

- AIX on the RS/6000 provides support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ("." vs "," for separating decimal fractions). There have been problems reported where the library linked with GCC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the LANG environment variable to "C" or "En_US".
- Even if you specify '-fdollars-in-identifiers', you cannot successfully use '$' in identifiers on the RS/6000 due to a restriction in the IBM assembler. GAS supports these identifiers.
- On the RS/6000, XLC version 1.3.0.0 will miscompile 'jump.c'. XLC version 1.3.0.1 or later fixes this problem. You can obtain XLC-1.3.0.2 by requesting PTF 421749 from IBM.
- There is an assembler bug in versions of DG/UX prior to 5.4.2.01 that occurs when the 'fldcr' instruction is used. GNU CC uses 'fldcr' on the 88100 to serialize volatile memory references. Use the option '-mno-serialize-volatile' if your version of the assembler has this bug.
- On VMS, GAS versions 1.38.1 and earlier may cause spurious warning messages from the linker. These warning messages complain of mismatched psect attributes. You can ignore them. See Section 5.5 “VMS Install,” page 133.
- On NewsOS version 3, if you include both of the files 'stddef.h' and 'sys/types.h', you get an error because there are two typedefs of size_t. You should change 'sys/types.h' by adding these lines around the definition of size_t:

```
#ifndef _SIZE_T
#define _SIZE_T
actual typedef here
#endif
s
```

- On the Alliant, the system's own convention for returning structures and unions is unusual, and is not compatible with GNU CC no matter what options are used.
- On the IBM RT PC, the MetaWare HighC compiler (hc) uses a different convention for structure and union returning. Use the option '`-mhc-struct-return`' to tell GNU CC to use a convention compatible with it.
- On Ultrix, the Fortran compiler expects registers 2 through 5 to be saved by function calls. However, the C compiler uses conventions compatible with BSD Unix: registers 2 through 5 may be clobbered by function calls.

  GNU CC uses the same convention as the Ultrix C compiler. You can use these options to produce code compatible with the Fortran compiler:

  ```
  -fcall-saved-r2 -fcall-saved-r3
  -fcall-saved-r4 -fcall-saved-r5
  ```
- On the WE32k, you may find that programs compiled with GNU CC do not work with the standard shared C ilbrary. You may need to link with the ordinary C compiler. If you do so, you must specify the following options:

  ```
  -L/usr/local/lib/gcc-lib/we32k-att-sysv/2.6.0 -lgcc -lc_s
  ```

  The first specifies where to find the library '`libgcc.a`' specified with the '`-lgcc`' option.

  GNU CC does linking by invoking `ld`, just as `cc` does, and there is no reason why it *should* matter which compilation program you use to invoke `ld`. If someone tracks this problem down, it can probably be fixed easily.
- On the Alpha, you may get assembler errors about invalid syntax as a result of floating point constants. This is due to a bug in the C library functions `ecvt`, `fcvt` and `gcvt`. Given valid floating point numbers, they sometimes print '`NaN`'.
- On Irix 4.0.5F (and perhaps in some other versions), an assembler bug sometimes reorders instructions incorrectly when optimization is turned on. If you think this may be happening to you, try using the GNU assembler; GAS version 2.1 supports ECOFF on Irix.

  Or use the '`-noasmopt`' option when you compile GNU CC with itself, and then again when you compile your program. (This is a temporary kludge to turn off assembler optimization on Irix.) If

this proves to be what you need, edit the assembler spec in the file 'specs' so that it unconditionally passes '-O0' to the assembler, and never passes '-O2' or '-O3'.

9.5 Problems Compiling Certain Programs

Certain programs have problems compiling.

- Parse errors may occur compiling X11 on a Decstation running Ultrix 4.2 because of problems in DEC's versions of the X11 header files 'X11/Xlib.h' and 'X11/Xutil.h'. People recommend adding '-I/usr/include/mit' to use the MIT versions of the header files, using the '-traditional' switch to turn off ANSI C, or fixing the header files by adding this:

  ```
  #ifdef __STDC__
  #define NeedFunctionPrototypes 0
  #endif
  ```

- If you have trouble compiling Perl on a SunOS 4 system, it may be because Perl specifies '-I/usr/ucbinclude'. This accesses the unfixed header files. Perl specifies the options

  ```
  -traditional -Dvolatile=__volatile__
  -I/usr/include/sun -I/usr/ucbinclude
  -fpcc-struct-return
  ```

  most of which are unnecessary with GCC 2.4.5 and newer versions. You can make a properly working Perl by setting ccflags to '-fwritable-strings' (implied by the '-traditional' in the original options) and cppflags to empty in 'config.sh', then typing './doSH; make depend; make'.

- On various 386 Unix systems derived from System V, including SCO, ISC, and ESIX, you may get error messages about running out of virtual memory while compiling certain programs.

  You can prevent this problem by linking GNU CC with the GNU malloc (which thus replaces the malloc that comes with the system). GNU malloc is available as a separate package, and also in the file 'src/gmalloc.c' in the GNU Emacs 19 distribution.

  If you have installed GNU malloc as a separate library package, use this option when you relink GNU CC:

  ```
  MALLOC=/usr/local/lib/libgmalloc.a
  ```

  Alternatively, if you have compiled 'gmalloc.c' from Emacs 19, copy the object file to 'gmalloc.o' and use this option when you relink GNU CC:

  ```
  MALLOC=gmalloc.o
  ```

9.6 Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The '`-traditional`' option eliminates many of these incompatibilities, *but not all*, by telling GNU C to behave like the other C compilers.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string or input. This is because `sscanf` incorrectly tries to write into the string constant. Likewise `fscanf` and `scanf`.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the '`-fwritable-strings`' flag, which directs GNU CC to handle string constants the same way most C compilers do. '`-traditional`' also has this effect, among others.

- `-2147483648` is positive.

  This is because 2147483648 cannot fit in the type `int`, so (following the ANSI C rules) its data type is `unsigned long int`. Negating this value yields 2147483648 again.

- GNU CC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) "a"
  ```

  will produce output `"a"` regardless of what the argument *a* is.

  The '`-traditional`' option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo ()
  {
    int a, b;
  ```

```
    a = fun1 ();
    if (setjmp (j))
      return a;

    a = fun2 ();
    /* longjmp (j) may occur in fun3. */
    return a + fun3 ();
  }
```

Here `a` may or may not be restored to its first value when the `longjmp` occurs. If `a` is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

If you use the '`-W`' option with the '`-O`' option, you will get a warning when GNU CC thinks such a problem might be possible.

The '`-traditional`' option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call `setjmp`. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GNU CC. For example, a program like this will not work:

  ```
  foobar (
  #define luser
          hack)
  ```

  ANSI C does not permit such a construct. It would make sense to support it when '`-traditional`' is used, but it is too much work to implement.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, a `extern` declaration affects all the rest of the file even if it happens within a block.

  The '`-traditional`' option directs GNU C to treat all `extern` declarations as global, like traditional compilers.

- In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the '`-traditional`' flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.

- PCC allows whitespace in the middle of compound assignment operators such as '+='. GNU CC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.
- GNU CC complains about unterminated character constants inside of preprocessing conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GNU CC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '/*. . .*/'. However, '-traditional' suppresses these error messages.
- Many user programs contain the declaration 'long time ();'. In the past, the system header files on many systems did not actually declare time, so it did not matter what type your program declared it to return. But in systems with ANSI C headers, time is declared to return time_t, and if that is not the same as long, then 'long time ();' is erroneous.

  The solution is to change your program to use time_t as the return type of time.
- When compiling functions that return float, PCC converts it to a double. GNU CC actually returns a float. If you are concerned with PCC compatibility, you should declare your functions to return double; you might as well say what you mean.
- When compiling functions that return structures or unions, GNU CC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros STRUCT_VALUE and STRUCT_INCOMING_VALUE tell GNU CC where to pass this address.

  By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the

place where the value is wanted. GNU CC does not use this method because it is slower and nonreentrant.

On some newer machines, PCC uses a reentrant convention for all structure and union returning. GNU CC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

You can tell GNU CC to use a compatible convention for all structure and union returning with the option '`-fpcc-struct-return`'.

- GNU C complains about program fragments such as '`0x74ae-0x4000`', which appear to be two hexadecimal constants separated by the minus operator. Actually, this string is a single *preprocessing token*. Each such token must correspond to one token in C. Since this does not, GNU C prints an error message. Although it may appear obvious that what is meant is an operator and two values, the ANSI C standard specifically requires that this be treated as erroneous.

  A *preprocessing token* is a *preprocessing number* if it begins with a digit and is followed by letters, underscores, digits, periods and '`e+`', '`e-`', '`E+`', or '`E-`' character sequences.

  To make the above program fragment valid, place whitespace in front of the minus sign. This whitespace will end the preprocessing number.

9.7 Fixed Header Files

GNU CC needs to install corrected versions of some system header files. This is because most target systems have some header files that won't work with GNU CC unless they are changed. Some have bugs, some are incompatible with ANSI C, and some depend on special features of other compilers.

Installing GNU CC automatically creates and installs the fixed header files, by running a program called `fixincludes` (or for certain targets an alternative such as `fixinc.svr4`). Normally, you don't need to pay attention to this. But there are cases where it doesn't do the right thing automatically.

- If you update the system's header files, such as by installing a new system version, the fixed header files of GNU CC are not automatically updated. The easiest way to update them is to reinstall GNU CC. (If you want to be clever, look in the makefile and you can find a shortcut.)
- On some systems, in particular SunOS 4, header file directories contain machine-specific symbolic links in certain places. This makes it possible to share most of the header files among hosts running the same version of SunOS 4 on different machine models.

The programs that fix the header files do not understand this special way of using symbolic links; therefore, the directory of fixed header files is good only for the machine model used to build it.

In SunOS 4, only programs that look inside the kernel will notice the difference between machine models. Therefore, for most purposes, you need not be concerned about this.

It is possible to make separate sets of fixed header files for the different machine models, and arrange a structure of symbolic links so as to use the proper set, but you'll have to do this by hand.

- On Lynxos, GNU CC by default does not fix the header files. This is because bugs in the shell cause the `fixincludes` script to fail.

  This means you will encounter problems due to bugs in the system header files. It may be no comfort that they aren't GNU CC's fault, but it does mean that there's nothing for us to do about them.

9.8 Disappointments and Misunderstandings

These problems are perhaps regrettable, but we don't know any practical way around them.

- Certain local variables aren't recognized by debuggers when you compile with optimization.

  This occurs because sometimes GNU CC optimizes the variable out of existence. There is no way to tell the debugger how to compute the value such a variable "would have had", and it is not clear that would be desirable anyway. So GNU CC simply does not mention the eliminated variable when it writes debugging information.

  You have to expect a certain amount of disagreement between the executable and your source code, when you use optimization.

- Users often think it is a bug when GNU CC reports an error for code like this:

```
int foo (struct mumble *);

struct mumble { ... };

int foo (struct mumble *x)
{ ... }
```

  This code really is erroneous, because the scope of `struct mumble` in the prototype is limited to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

But in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

This behavior may seem silly, but it's what the ANSI standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype. It's not worth being incompatible with ANSI C just to avoid an error for the example shown above.

- Accesses to bitfields even in volatile objects works by accessing larger objects, such as a byte or a word. You cannot rely on what size of object is accessed in order to read or write the bitfield; it may even vary for a given bitfield according to the precise usage.

  If you care about controlling the amount of memory that is accessed, use volatile but do not use bitfields.

- GNU CC comes with shell scripts to fix certain known problems in system header files. They install corrected copies of various header files in a special directory where only GNU CC will normally look for them. The scripts adapt to various systems by searching all the system header files for the problem cases that we know about.

  If new system header files are installed, nothing automatically arranges to update the corrected header files. You will have to reinstall GNU CC to fix the new header files. More specifically, go to the build directory and delete the files '`stmp-fixinc`' and '`stmp-headers`', and the subdirectory `include`; then do '`make install`' again.

- On 68000 systems, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the the floating point registers hold a few more bits of precision than fit in a `double` in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

  You can partially avoid this problem by using the '`-ffloat-store`' option (see Section 4.8 "Optimize Options," page 53).

- On the MIPS, variable argument functions using '`varargs.h`' cannot have a floating point value for the first argument. The reason for this is that in the absence of a prototype in scope, if the first argument is a floating point, it is passed in a floating point register, rather than an integer register.

  If the code is rewritten to use the ANSI standard '`stdarg.h`' method of variable arguments, and the prototype is in scope at the time of the call, everything will work fine.

9.9 Common Misunderstandings with GNU C++

C++ is a complex language and an evolving one, and its standard definition (the ANSI C++ draft standard) is also evolving. As a result, your C++ compiler may occasionally surprise you, even when its behavior is correct. This section discusses some areas that frequently give rise to questions of this sort.

9.9.1 Declare *and* Define Static Members

When a class has static data members, it is not enough to *declare* the static member; you must also *define* it. For example:

```
class Foo
{
  ...
  void method();
  static int bar;
};
```

This declaration only establishes that the class `Foo` has an `int` named `Foo::bar`, and a member function named `Foo::method`. But you still need to define *both* `method` and `bar` elsewhere. According to the draft ANSI standard, you must supply an initializer in one (and only one) source file, such as:

```
int Foo::bar = 0;
```

Other C++ compilers may not correctly implement the standard behavior. As a result, when you switch to `g++` from one of these compilers, you may discover that a program that appeared to work correctly in fact does not conform to the standard: `g++` reports as undefined symbols any static data members that lack definitions.

9.9.2 Temporaries May Vanish Before You Expect

It is dangerous to use pointers or references to *portions* of a temporary object. The compiler may very well delete the object before you expect it to, leaving a pointer to garbage. The most common place where this problem crops up is in classes like the libg++ `String` class, that define a conversion function to type `char *` or `const char *`. However, any class that returns a pointer to some internal structure is potentially subject to this problem.

For example, a program may use a function `strfunc` that returns `String` objects, and another function `charfunc` that operates on pointers to `char`:

```
String strfunc ();
void charfunc (const char *);
```

In this situation, it may seem natural to write '`charfunc (strfunc ());`' based on the knowledge that class `String` has an explicit conversion to `char` pointers. However, what really happens is akin to '`charfunc (strfunc ().convert ());`', where the `convert` method is a function to do the same data conversion normally performed by a cast. Since the last use of the temporary `String` object is the call to the conversion function, the compiler may delete that object before actually calling `charfunc`. The compiler has no way of knowing that deleting the `String` object will invalidate the pointer. The pointer then points to garbage, so that by the time `charfunc` is called, it gets an invalid argument.

Code like this may run successfully under some other compilers, especially those that delete temporaries relatively late. However, the GNU C++ behavior is also standard-conformant, so if your program depends on late destruction of temporaries it is not portable.

If you think this is surprising, you should be aware that the ANSI C++ committee continues to debate the lifetime-of-temporaries problem.

For now, at least, the safe way to write such code is to give the temporary a name, which forces it to remain until the end of the scope of the name. For example:

```
String& tmp = strfunc ();
charfunc (tmp);
```

9.10 Caveats of using `protoize`

The conversion programs `protoize` and `unprotoize` can sometimes change a source file in a way that won't work unless you rearrange it.

- `protoize` can insert references to a type name or type tag before the definition, or in a file where they are not defined.

  If this happens, compiler error messages should show you where the new references are, so fixing the file by hand is straightforward.
- There are some C constructs which `protoize` cannot figure out. For example, it can't determine argument types for declaring a pointer-to-function variable; this you must do by hand. `protoize` inserts a comment containing '`???`' each time it finds such a variable; so you can find all such variables by searching for this string. ANSI C does not require declaring the argument types of pointer-to-function types.
- Using `unprotoize` can easily introduce bugs. If the program relied on prototypes to bring about conversion of arguments, these conversions will not take place in the program without prototypes. One case in which you can be sure `unprotoize` is safe is when you are removing prototypes that were made with `protoize`; if the program

worked before without any prototypes, it will work again without them.

You can find all the places where this problem might occur by compiling the program with the '-Wconversion' option. It prints a warning whenever an argument is converted.

- Both conversion programs can be confused if there are macro calls in and around the text to be converted. In other words, the standard syntax for a declaration or definition must not result from expanding a macro. This problem is inherent in the design of C and cannot be fixed. If only a few functions have confusing macro calls, you can easily convert them manually.
- `protoize` cannot get the argument types for a function whose definition was not actually compiled due to preprocessing conditionals. When this happens, `protoize` changes nothing in regard to such a function. `protoize` tries to detect such instances and warn about them.

  You can generally work around this problem by using `protoize` step by step, each time specifying a different set of '-D' options for compilation, until all of the functions have been converted. There is no automatic way to verify that you have got them all, however.
- Confusion may result if there is an occasion to convert a function declaration or definition in a region of source code where there is more than one formal parameter list present. Thus, attempts to convert code containing multiple (conditionally compiled) versions of a single function header (in the same vicinity) may not produce the desired (or expected) results.

  If you plan on converting source files which contain such code, it is recommended that you first make sure that each conditionally compiled region of source code which contains an alternative function header also contains at least one additional follower token (past the final right parenthesis of the function header). This should circumvent the problem.
- `unprotoize` can become confused when trying to convert a function definition or declaration which contains a declaration for a pointer-to-function formal argument which has the same name as the function being defined or declared. We recommand you avoid such choices of formal parameter names.
- You might also want to correct some of the indentation by hand and break long lines. (The conversion programs don't write lines longer than eighty characters in any case.)

9.11 Certain Changes We Don't Want to Make

This section lists changes that people frequently request, but which we do not make because we think GNU CC is better without them.

- Checking the number and type of arguments to a function which has an old-fashioned definition and no prototype.

  Such a feature would work only occasionally—only for calls that appear in the same file as the called function, following the definition. The only way to check all calls reliably is to add a prototype for the function. But adding a prototype eliminates the motivation for this feature. So the feature is not worthwhile.
- Warning about using an expression whose type is signed as a shift count.

  Shift count operands are probably signed more often than unsigned. Warning about this would cause far more annoyance than good.
- Warning about assigning a signed value to an unsigned variable.

  Such assignments must be very common; warning about them would cause more annoyance than good.
- Warning about unreachable code.

  It's very common to have unreachable code in machine-generated programs. For example, this happens normally in some files of GNU C itself.
- Warning when a non-void function value is ignored.

  Coming as I do from a Lisp background, I balk at the idea that there is something dangerous about discarding a value. There are functions that return values which some callers may find useful; it makes no sense to clutter the program with a cast to `void` whenever the value isn't useful.
- Assuming (for optimization) that the address of an external symbol is never zero.

  This assumption is false on certain systems when '`#pragma weak`' is used.
- Making '`-fshort-enums`' the default.

  This would cause storage layout to be incompatible with most other C compilers. And it doesn't seem very important, given that you can get the same result in other ways. The case where it matters most is when the enumeration-valued object is inside a structure, and in that case you can specify a field width explicitly.
- Making bitfields unsigned by default on particular machines where "the ABI standard" says to do so.

The ANSI C standard leaves it up to the implementation whether a bitfield declared plain `int` is signed or not. This in effect creates two alternative dialects of C.

The GNU C compiler supports both dialects; you can specify the signed dialect with '`-fsigned-bitfields`' and the unsigned dialect with '`-funsigned-bitfields`'. However, this leaves open the question of which dialect to use by default.

Currently, the preferred dialect makes plain bitfields signed, because this is simplest. Since `int` is the same as `signed int` in every other context, it is cleanest for them to be the same in bitfields as well.

Some computer manufacturers have published Application Binary Interface standards which specify that plain bitfields should be unsigned. It is a mistake, however, to say anything about this issue in an ABI. This is because the handling of plain bitfields distinguishes two dialects of C. Both dialects are meaningful on every type of machine. Whether a particular object file was compiled using signed bitfields or unsigned is of no concern to other object files, even if they access the same bitfields in the same data structures.

A given program is written in one or the other of these two dialects. The program stands a chance to work on most any machine if it is compiled with the proper dialect. It is unlikely to work at all if compiled with the wrong dialect.

Many users appreciate the GNU C compiler because it provides an environment that is uniform across machines. These users would be inconvenienced if the compiler treated plain bitfields differently on certain machines.

Occasionally users write programs intended only for a particular machine type. On these occasions, the users would benefit if the GNU C compiler were to support by default the same dialect as the other compilers on that machine. But such applications are rare. And users writing a program to run on more than one type of machine cannot possibly benefit from this kind of compatibility.

This is why GNU CC does and will treat plain bitfields in the same fashion on all types of machines (by default).

There are some arguments for making bitfields unsigned by default on all machines. If, for example, this becomes a universal de facto standard, it would make sense for GNU CC to go along with it. This is something to be considered in the future.

(Of course, users strongly concerned about portability should indicate explicitly in each bitfield whether it is signed or not. In this way, they write programs which have the same meaning in both C dialects.)

- Undefining `__STDC__` when '`-ansi`' is not used.

  Currently, GNU CC defines `__STDC__` as long as you don't use '`-traditional`'. This provides good results in practice.

  Programmers normally use conditionals on `__STDC__` to ask whether it is safe to use certain features of ANSI C, such as function prototypes or ANSI token concatenation. Since plain '`gcc`' supports all the features of ANSI C, the correct answer to these questions is "yes".

  Some users try to use `__STDC__` to check for the availability of certain library facilities. This is actually incorrect usage in an ANSI C program, because the ANSI C standard says that a conforming freestanding implementation should define `__STDC__` even though it does not have the library facilities. '`gcc -ansi -pedantic`' is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ANSI C library.

  Sometimes people say that defining `__STDC__` in a compiler that does not completely conform to the ANSI C standard somehow violates the standard. This is illogical. The standard is a standard for compilers that claim to support ANSI C, such as '`gcc -ansi`'—not for other compilers such as plain '`gcc`'. Whatever the ANSI C standard says is relevant to the design of plain '`gcc`' without '`-ansi`' only for pragmatic reasons, not as a requirement.

- Undefining `__STDC__` in C++.

  Programs written to compile with C++-to-C translators get the value of `__STDC__` that goes with the C compiler that is subsequently used. These programs must test `__STDC__` to determine what kind of C preprocessor that compiler uses: whether they should concatenate tokens in the ANSI C fashion or in the traditional fashion.

  These programs work properly with GNU C++ if `__STDC__` is defined. They would not work otherwise.

  In addition, many header files are written to provide prototypes in ANSI C but not in traditional C. Many of these header files can work without change in C++ provided `__STDC__` is defined. If `__STDC__` is not defined, they will all fail, and will all need to be changed to test explicitly for C++ as well.

- Deleting "empty" loops.

  GNU CC does not delete "empty" loops because the most likely reason you would put one in a program is to have a delay. Deleting them will not make real programs run any faster, so it would be pointless.

  It would be different if optimization of a nonempty loop could produce an empty one. But this generally can't happen.

- Making side effects happen in the same order as in some other compiler.

  It is never safe to depend on the order of evaluation of side effects. For example, a function call like this may very well behave differently from one compiler to another:

  ```
  void func (int, int);

  int i = 2;
  func (i++, i++);
  ```

  There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. `func` might get the arguments '`3, 4`', or it might get '`4, 3`', or even '`3, 3`'.
- Not allowing structures with volatile fields in registers.

  Strictly speaking, there is no prohibition in the ANSI C standard against allowing structures with volatile fields in registers, but it does not seem to make any sense and is probably not what you wanted to do. So the compiler will give an error message in this case.

9.12 Warning Messages and Error Messages

The GNU compiler can produce two kinds of diagnostics: errors and warnings. Each kind has a different purpose:

> *Errors* report problems that make it impossible to compile your program. GNU CC reports errors with the source file name and line number where the problem is apparent.
>
> *Warnings* report other unusual conditions in your code that *may* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text '`warning:`' to distinguish them from error messages.

Warnings may indicate danger points where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of nonstandard features of GNU C or C++. Many warnings are issued only if you ask for them, with one of the '`-W`' options (for instance, '`-Wall`' requests a variety of useful warnings).

GNU CC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The '`-pedantic`' option tells GNU CC to issue warnings in such cases; '`-pedantic-errors`'

says to make them errors instead. This does not mean that *all* non-ANSI constructs get warnings or errors.

See Section 4.6 “Options to Request or Suppress Warnings,” page 40, for more detail on these and related command-line options.

10 Reporting Bugs

Your bug reports play an essential role in making GNU CC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 9 "Trouble," page 205. If it isn't known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. (If it does not, look in the service directory; see Chapter 11 "Service," page 243.) In any case, the principal function of a bug report is to help the entire community by making the next version of GNU CC work better. Bug reports are your contribution to the maintenance of GNU CC.

Since the maintainers are very overloaded, we cannot respond to every bug report. However, if the bug has not been fixed, we are likely to send you a patch and ask you to tell us whether it works.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

10.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have run into an incompatibility between GNU C and traditional C (see Section 9.6 "Incompatibilities," page 218). These incompatibilities might be considered bugs, but they are inescapable consequences of valuable features.

  Or you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C or C++ compiler.

  For example, in many nonoptimizing compilers, you can write '`x;`' at the end of a function instead of '`return x;`', with the same results.

But the value of the function is undefined if `return` is omitted; it is not a bug when GNU CC produces different results.

Problems often result from expressions with two increment operators, as in `f (*p++, *p++)`. Your previous compiler might have interpreted that expression the way you intended; GNU CC might interpret it another way. Neither compiler is wrong. The bug is in your code.

After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.

- If the compiler produces an error message for valid input, that is a compiler bug.
- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".
- If you are an experienced user of C or C++ compilers, your suggestions for improvement of GNU CC or GNU C++ are welcome in any case.

10.2 Where to Report Bugs

Send bug reports for GNU C to '`bug-gcc@prep.ai.mit.edu`'.

Send bug reports for GNU C++ to '`bug-g++@prep.ai.mit.edu`'. If your bug involves the C++ class library libg++, send mail to '`bug-lib-g++@prep.ai.mit.edu`'. If you're not sure, you can send the bug report to both lists.

Do not send bug reports to '`help-gcc@prep.ai.mit.edu`**' or to the newsgroup '**`gnu.gcc.help`**'.** Most users of GNU CC do not want to receive bug reports. Those that do, have asked to be on '`bug-gcc`' and/or '`bug-g++`'.

The mailing lists '`bug-gcc`' and '`bug-g++`' both have newsgroups which serve as repeaters: '`gnu.gcc.bug`' and '`gnu.g++.bug`'. Each mailing list and its 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 does not contain a mail path back to the sender. Thus, if maintainers need more information, they may be unable to reach you. For this reason, you should always send bug reports by mail to the proper mailing list.

As a last resort, send bug reports on paper to:

```
GNU Compiler Bugs
Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
```

10.3 How to Report Bugs

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 they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, 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 compiler 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 someone to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.

Do not compress and encode any part of your bug report using programs such as 'uuencode'. If you do so it will slow down the processing of your bug. If you must submit multiple large files, use 'shar', which allows us to read your message without having to run any decompression programs.

To enable someone to investigate the bug, you should include all these things:

- The version of GNU CC. You can get this by running it with the '-v' option.

Without this, we won't know whether there is any point in looking for the bug in the current version of GNU CC.

- A complete input file that will reproduce the bug. If the bug is in the C preprocessor, send a source file and any header files that it requires. If the bug is in the compiler proper ('`cc1`'), run your source file through the C preprocessor by doing '`gcc -E` *`sourcefile`* `>` *`outfile`*', then include the contents of *`outfile`* in the bug report. (When you do this, use the same '`-I`', '`-D`' or '`-U`' options that you used in actual compilation.)

  A single statement is not enough of an example. In order to compile it, it must be embedded in a complete file of compiler input; and the bug might depend on the details of how this is done.

  Without a real example one can compile, all anyone can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function they happen in.

  Even if the input file that fails comes from a GNU program, you should still send the complete test case. Don't ask the GNU CC maintainers to do the extra work of obtaining the program in question—they are all overworked as it is. Also, the problem may depend on what is in the header files on your system; it is unreliable for the GNU CC maintainers to try the problem with the header files available to them. By sending CPP output, you can eliminate this source of uncertainty and save us a certain percentage of wild goose chases.

- The command arguments you gave GNU CC or GNU C++ to compile that example and observe the bug. For example, did you use '`-O`'? To guarantee you won't omit something important, list all the options.

  If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.

- The type of machine you are using, and the operating system name and version number.
- The operands you gave to the `configure` command when you installed the compiler.
- A complete list of any modifications you have made to the compiler source. (We don't promise to investigate the bug unless it happens in an unmodified compiler. But if you've made modifications and don't tell us, then you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough—send a context diff for them.

  Adding files of your own (such as a machine description for a machine we don't support) is a modification of the compiler source.

- Details of any other deviations from the standard procedure for installing GNU CC.
- A description of what behavior you observe that you believe is incorrect. For example, "The compiler gets a fatal signal," or, "The assembler instruction at line 208 in the output is incorrect."

  Of course, if the bug is that the compiler gets a fatal signal, then one can't miss it. But if the bug is incorrect output, the maintainer might not notice unless it is glaringly wrong. None of us has time to study all the assembler code from a 50-line C program just on the chance that one instruction might be wrong. We need *you* to do this part!

  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 the compiler 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 the copy here would not. If you *said* to expect a crash, then when the compiler here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

  If the problem is a diagnostic when compiling GNU CC with some other compiler, say whether it is a warning or an error.

  Often the observed symptom is incorrect output when your program is run. Sad to say, this is not enough information unless the program is short and simple. None of us has time to study a large program to figure out how it would work if compiled correctly, much less which line of it was compiled wrong. So you will have to do that. Tell us which source line it is, and what incorrect result happens when that line is executed. A person who understands the program can find this as easily as finding a bug in the program itself.
- If you send examples of assembler code output from GNU CC or GNU C++, please use '`-g`' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.
- If you wish to mention something in the GNU CC source, refer to it by context, not by line number.

  The line numbers in the development sources don't match those in your sources. Your line numbers would convey no useful information to the maintainers.
- Additional information from a debugger might enable someone to find a problem on a machine which he does not have available. However, you need to think when you collect this information if you want it to have any chance of being useful.

For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about GNU CC because the compiler is largely data-driven; the same functions are called over and over for different RTL insns, doing different things depending on the details of the insn.

Most of the arguments listed in the backtrace are useless because they are pointers to RTL list structure. The numeric values of the pointers, which the debugger prints in the backtrace, have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are other such pointers).

In addition, most compiler passes consist of one or more loops that scan the RTL insn sequence. The most vital piece of information about such a loop—which insn it has reached—is usually in a local variable, not in an argument.

What you need to provide in addition to a backtrace are the values of the local variables for several stack frames up. When a local variable or an argument is an RTX, first print its value and then use the GDB command `pr` to print the RTL expression that it points to. (If GDB doesn't run on your machine, use your debugger to call the function `debug_rtx` with the RTX as an argument.) In general, whenever a variable is a pointer, its value is no use without the data it points to.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well 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. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most GNU CC bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations if the crucial function depends on them. (Exception: inline functions may affect compilation of functions defined later in the file.)

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.

- In particular, some people insert conditionals '`#ifdef BUG`' around a statement which, if removed, makes the bug not happen. These are just clutter; we won't pay any attention to them anyway. Besides, you should send us cpp output, and that can't have conditionals.
- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't 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 GNU CC it is very hard to construct an example that will make the program follow a certain path through the code. If you don't send the example, we won't be able to construct one, so we won't be able to verify that the bug is fixed.

  And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we won't install it. A test case will help us to understand.

  See Section 10.4 "Sending Patches," page 239, for guidelines on how to make it easy for us to understand and install your patches.
- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even I can't guess right about such things without first using the debugger to find the facts.
- A core dump file.

  We have no way of examining a core dump for your type of machine unless we have an identical system—and if we do have one, we should be able to reproduce the crash ourselves.

10.4 Sending Patches for GNU CC

If you would like to write bug fixes or improvements for the GNU C compiler, that is very helpful. Send suggested fixes to the bug report mailing list, `bug-gcc@prep.ai.mit.edu`.

Please follow these guidelines so we can study your patches efficiently. If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU C is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.

(Referring to a bug report is not as good as including it, because then we will have to look it up, and we have probably already deleted it if we've already fixed the bug.)

- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is right, we might have trouble judging it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them—to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

  If you send each change as soon as you have written it, with its own explanation, then the two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.

  Ideally, each change you send should be impossible to subdivide into parts that we might want to consider separately, because each of its parts gets its motivation from the other parts.
- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

  Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.
- Use '`diff -c`' to make your diffs. Diffs without context are hard for us to install reliably. More than that, they make it hard for us to study the diffs to decide whether we want to install them. Unidiff format is better than contextless diffs, but not as easy to read as '`-c`' format.

  If you have GNU diff, use '`diff -cp`', which shows the name of the function that each change occurs in.
- Write the change log entries for your changes. We get lots of changes, and we don't have time to do all the change log writing ourselves.

  Read the '`ChangeLog`' file to see what sorts of information to put in, and to learn the style that we use. The purpose of the change log

is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but the explanation will be much more useful if you put it in comments in the code.

If you would like your name to appear in the header line for who made the change, send us the header line.

- When you write the fix, keep in mind that we can't install a change that would break other systems.

  People often suggest fixing a problem by changing machine-independent files such as '`toplev.c`' to do something special that a particular system needs. Sometimes it is totally obvious that such changes would break GNU CC for almost all users. We can't possibly make a change like that. At best it might tell us how to write another patch that would solve the problem acceptably.

  Sometimes people send fixes that *might* be an improvement in general—but it is hard to be sure of this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

  The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

  Please help us keep up with the workload by designing the patch in a form that is good to install.

11 How To Get Help with GNU CC

If you need help installing, using or changing GNU CC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `bug-gcc@prep.ai.mit.edu`, and if that brings no response, try `help-gcc@prep.ai.mit.edu`.
- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named 'SERVICE' in the GNU CC distribution.

12 Using GNU CC on VMS

Here is how to use GNU CC on VMS.

12.1 Include Files and VMS

Due to the differences between the filesystems of Unix and VMS, GNU CC attempts to translate file names in '#include' into names that VMS will understand. The basic strategy is to prepend a prefix to the specification of the include file, convert the whole filename to a VMS filename, and then try to open the file. GNU CC tries various prefixes one by one until one of them succeeds:

1. The first prefix is the 'GNU_CC_INCLUDE:' logical name: this is where GNU C header files are traditionally stored. If you wish to store header files in non-standard locations, then you can assign the logical 'GNU_CC_INCLUDE' to be a search list, where each element of the list is suitable for use with a rooted logical.
2. The next prefix tried is 'SYS$SYSROOT:[SYSLIB.]'. This is where VAX-C header files are traditionally stored.
3. If the include file specification by itself is a valid VMS filename, the preprocessor then uses this name with no prefix in an attempt to open the include file.
4. If the file specification is not a valid VMS filename (i.e. does not contain a device or a directory specifier, and contains a '/' character), the preprocessor tries to convert it from Unix syntax to VMS syntax.

   Conversion works like this: the first directory name becomes a device, and the rest of the directories are converted into VMS-format directory names. For example, the name 'X11/foobar.h' is translated to 'X11:[000000]foobar.h' or 'X11:foobar.h', whichever one can be opened. This strategy allows you to assign a logical name to point to the actual location of the header files.
5. If none of these strategies succeeds, the '#include' fails.

Include directives of the form:

```
#include foobar
```

are a common source of incompatibility between VAX-C and GNU CC. VAX-C treats this much like a standard `#include <foobar.h>` directive. That is incompatible with the ANSI C behavior implemented by GNU CC: to expand the name `foobar` as a macro. Macro expansion should eventually yield one of the two standard formats for `#include`:

```
#include "file"
#include <file>
```

If you have this problem, the best solution is to modify the source to convert the `#include` directives to one of the two standard forms. That will work with either compiler. If you want a quick and dirty fix, define the file names as macros with the proper expansion, like this:

```
#define stdio <stdio.h>
```

This will work, as long as the name doesn't conflict with anything else in the program.

Another source of incompatibility is that VAX-C assumes that:

```
#include "foobar"
```

is actually asking for the file 'foobar.h'. GNU CC does not make this assumption, and instead takes what you ask for literally; it tries to read the file 'foobar'. The best way to avoid this problem is to always specify the desired file extension in your include directives.

GNU CC for VMS is distributed with a set of include files that is sufficient to compile most general purpose programs. Even though the GNU CC distribution does not contain header files to define constants and structures for some VMS system-specific functions, there is no reason why you cannot use GNU CC with any of these functions. You first may have to generate or create header files, either by using the public domain utility `UNSDL` (which can be found on a DECUS tape), or by extracting the relevant modules from one of the system macro libraries, and using an editor to construct a C header file.

A `#include` file name cannot contain a DECNET node name. The preprocessor reports an I/O error if you attempt to use a node name, whether explicitly, or implicitly via a logical name.

12.2 Global Declarations and VMS

GNU CC does not provide the `globalref`, `globaldef` and `globalvalue` keywords of VAX-C. You can get the same effect with an obscure feature of GAS, the GNU assembler. (This requires GAS version 1.39 or later.) The following macros allow you to use this feature in a fairly natural way:

```
#ifdef __GNUC__
#define GLOBALREF(TYPE,NAME)                        \
  TYPE NAME                                         \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME)
#define GLOBALDEF(TYPE,NAME,VALUE)                  \
  TYPE NAME                                         \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME) \
    = VALUE
#define GLOBALVALUEREF(TYPE,NAME)                   \
  const TYPE NAME[1]                                \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)
```

```
#define GLOBALVALUEDEF(TYPE,NAME,VALUE)            \
  const TYPE NAME[1]                                \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)  \
    = {VALUE}
#else
#define GLOBALREF(TYPE,NAME) \
  globalref TYPE NAME
#define GLOBALDEF(TYPE,NAME,VALUE) \
  globaldef TYPE NAME = VALUE
#define GLOBALVALUEDEF(TYPE,NAME,VALUE) \
  globalvalue TYPE NAME = VALUE
#define GLOBALVALUEREF(TYPE,NAME) \
  globalvalue TYPE NAME
#endif
```

(The `_$$PsectAttributes_GLOBALSYMBOL` prefix at the start of the name is removed by the assembler, after it has modified the attributes of the symbol). These macros are provided in the VMS binaries distribution in a header file 'GNU_HACKS.H'. An example of the usage is:

```
GLOBALREF (int, ijk);
GLOBALDEF (int, jkl, 0);
```

The macros `GLOBALREF` and `GLOBALDEF` cannot be used straightforwardly for arrays, since there is no way to insert the array dimension into the declaration at the right place. However, you can declare an array with these macros if you first define a typedef for the array type, like this:

```
typedef int intvector[10];
GLOBALREF (intvector, foo);
```

Array and structure initializers will also break the macros; you can define the initializer to be a macro of its own, or you can expand the `GLOBALDEF` macro by hand. You may find a case where you wish to use the `GLOBALDEF` macro with a large array, but you are not interested in explicitly initializing each element of the array. In such cases you can use an initializer like: `{0,}`, which will initialize the entire array to `0`.

A shortcoming of this implementation is that a variable declared with `GLOBALVALUEREF` or `GLOBALVALUEDEF` is always an array. For example, the declaration:

```
GLOBALVALUEREF(int, ijk);
```

declares the variable `ijk` as an array of type `int [1]`. This is done because a globalvalue is actually a constant; its "value" is what the linker would normally consider an address. That is not how an integer value works in C, but it is how an array works. So treating the symbol as an array name gives consistent results—with the exception that the value seems to have the wrong type. **Don't try to access an element of the array.** It doesn't have any elements. The array "address" may not be the address of actual storage.

The fact that the symbol is an array may lead to warnings where the variable is used. Insert type casts to avoid the warnings. Here is an example; it takes advantage of the ANSI C feature allowing macros that expand to use the same name as the macro itself.

```
GLOBALVALUEREF (int, ss$_normal);
GLOBALVALUEDEF (int, xyzzy,123);
#ifdef __GNUC__
#define ss$_normal ((int) ss$_normal)
#define xyzzy ((int) xyzzy)
#endif
```

Don't use `globaldef` or `globalref` with a variable whose type is an enumeration type; this is not implemented. Instead, make the variable an integer, and use a `globalvaluedef` for each of the enumeration values. An example of this would be:

```
#ifdef __GNUC__
GLOBALDEF (int, color, 0);
GLOBALVALUEDEF (int, RED, 0);
GLOBALVALUEDEF (int, BLUE, 1);
GLOBALVALUEDEF (int, GREEN, 3);
#else
enum globaldef color {RED, BLUE, GREEN = 3};
#endif
```

12.3 Other VMS Issues

GNU CC automatically arranges for `main` to return 1 by default if you fail to specify an explicit return value. This will be interpreted by VMS as a status code indicating a normal successful completion. Version 1 of GNU CC did not provide this default.

GNU CC on VMS works only with the GNU assembler, GAS. You need version 1.37 or later of GAS in order to produce value debugging information for the VMS debugger. Use the ordinary VMS linker with the object files produced by GAS.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked to the sharable '`VAXCRTL`' library. Now this should work.

A caveat for use of `const` global variables: the `const` modifier must be specified in every external declaration of the variable in all of the source files that use that variable. Otherwise the linker will issue warnings about conflicting attributes for the variable. Your program will still work despite the warnings, but the variable will be placed in writable storage.

Although the VMS linker does distinguish between upper and lower case letters in global symbols, most VMS compilers convert all such symbols into upper case and most run-time library routines also have

upper case names. To be able to reliably call such routines, GNU CC (by means of the assembler GAS) converts global symbols into upper case like other VMS compilers. However, since the usual practice in C is to distinguish case, GNU CC (via GAS) tries to preserve usual C behavior by augmenting each name that is not all lower case. This means truncating the name to at most 23 characters and then adding more characters at the end which encode the case pattern of those 23. Names which contain at least one dollar sign are an exception; they are converted directly into upper case without augmentation.

Name augmentation yields bad results for programs that use precompiled libraries (such as Xlib) which were generated by another compiler. You can use the compiler option '/NOCASE_HACK' to inhibit augmentation; it makes external C functions and variables case-independent as is usual on VMS. Alternatively, you could write all references to the functions and variables in such libraries using lower case; this will work on VMS, but is not portable to other systems. The compiler option '/NAMES' also provides control over global name handling.

Function and variable names are handled somewhat differently with GNU C++. The GNU C++ compiler performs *name mangling* on function names, which means that it adds information to the function name to describe the data types of the arguments that the function takes. One result of this is that the name of a function can become very long. Since the VMS linker only recognizes the first 31 characters in a name, special action is taken to ensure that each function and variable has a unique name that can be represented in 31 characters.

If the name (plus a name augmentation, if required) is less than 32 characters in length, then no special action is performed. If the name is longer than 31 characters, the assembler (GAS) will generate a hash string based upon the function name, truncate the function name to 23 characters, and append the hash string to the truncated name. If the '/VERBOSE' compiler option is used, the assembler will print both the full and truncated names of each symbol that is truncated.

The '/NOCASE_HACK' compiler option should not be used when you are compiling programs that use libg++. libg++ has several instances of objects (i.e. `Filebuf` and `filebuf`) which become indistinguishable in a case-insensitive environment. This leads to cases where you need to inhibit augmentation selectively (if you were using libg++ and Xlib in the same program, for example). There is no special feature for doing this, but you can get the result by defining a macro for each mixed case symbol for which you wish to inhibit augmentation. The macro should expand into the lower case equivalent of itself. For example:

```
#define StuDlyCapS studlycaps
```

These macro definitions can be placed in a header file to minimize the number of changes to your source code.

Index

!

‘!’ in constraint 177

#

‘#’ in constraint 178
`#pragma implementation`, implied 192
`#pragma`, reason for not using 160

$

`$` 162

%

‘%’ in constraint 178

&

‘&’ in constraint 178

’

’ 220

-

`-lgcc`, use with `-nostdlib` 62
`-nostdlib` and unresolved references 62

=

‘=’ in constraint 178

?

‘?’ in constraint 177
`?:` extensions 148, 149
?: side effect 149

_

‘_’ in variables in macros 147
`__builtin_apply` 146
`__builtin_apply_args` 146
`__builtin_return` 146
`__main` 137

+

‘+’ in constraint 178

>

‘>’ in constraint 175
`>?` 191

<

‘<’ in constraint 175
`<?` 191

0

‘0’ in constraint 176

A

`abort` 32
`abs` 32
address constraints 176
address of a label 143
`address_operand` 177
`aligned` attribute 163, 167
alignment 162
Alliant 216
`alloca` 32
`alloca` and SunOs 110
`alloca` vs variable-length arrays . . . 151
`alloca`, for SunOs 133
`alloca`, for Unos 120
alternate keywords 187
AMD29K options 73
ANSI support 31
apostrophes 220
arguments in frame (88k) 76
ARM options 74
arrays of length zero 151
arrays of variable length 151
arrays, non-lvalue 153
`asm` constraints 174
`asm` expressions 170
assembler instructions 170
assembler names for identifiers 184
assembler syntax, 88k 77

assembly code, invalid . . . 233
attribute of types . . . 166
attribute of variables . . . 163
autoincrement/decrement addressing . . . 174
automatic `inline` for C++ member fns . . . 169

B

backtrace for bug reports . . . 237
Bison parser generator . . . 107
bit shift overflow (88k) . . . 78
bug criteria . . . 233
bug report mailing lists . . . 234
bugs . . . 233
bugs, known . . . 205
builtin functions . . . 32
byte writes (29k) . . . 73

C

C compilation options . . . 23
C intermediate output, nonexistent . . . 21
C language extensions . . . 141
C language, traditional . . . 32
`C_INCLUDE_PATH` . . . 99
`c++` . . . 30
C++ . . . 21
C++ compilation options . . . 23
C++ interface and implementation headers . . . 191
C++ language extensions . . . 189
C++ member fns, automatically `inline` . . . 169
C++ misunderstandings . . . 224
C++ named return value . . . 189
C++ options, command line . . . 35
C++ pragmas, effect on inlining . . . 193
C++ signatures . . . 196
C++ source file suffixes . . . 30
C++ static data, declaring and defining . . . 224
C++ subtype polymorphism . . . 196
C++ type abstraction . . . 196
calling conventions for interrupts . . . 161
case labels in initializers . . . 155
case ranges . . . 156
case sensitivity and VMS . . . 248
cast to a union . . . 157
casts as lvalues . . . 148
code generation conventions . . . 94
command options . . . 23
compilation in a separate directory . . . 127
compiler bugs, reporting . . . 235
compiler compared to C++ preprocessor . . . 21
compiler options, C++ . . . 35
compiler version, specifying . . . 65
`COMPILER_PATH` . . . 99
complex numbers . . . 150
compound expressions as lvalues . . . 148
computed gotos . . . 143
conditional expressions as lvalues . . . 148
conditional expressions, extensions . . . 149
configurations supported by GNU CC . . . 111
conflicting types . . . 222
`const` applied to function . . . 157
`const` function attribute . . . 158
constants in constraints . . . 175
constraint modifier characters . . . 177
constraint, matching . . . 176
constraints, `asm` . . . 174
constraints, machine specific . . . 178
constructing calls . . . 146
constructor expressions . . . 154
`constructor` function attribute . . . 160
constructors vs `goto` . . . 191
constructors, automatic calls . . . 137
contributors . . . 11
Convex options . . . 72
core dump . . . 233
`cos` . . . 32
`CPLUS_INCLUDE_PATH` . . . 99
cross compiling . . . 65
cross-compiler, installation . . . 127

D

'`d`' in constraint . . . 175
DBX . . . 212
deallocating variable length arrays . . . 151
`debug_rtx` . . . 238
debugging information options . . . 48
debugging, 88k OCS . . . 75
declaration scope . . . 219
declarations inside expressions . . . 141
declaring attributes of functions . . . 157

declaring static data in C++ 224
default implementation, signature member function 197
defining static data in C++. 224
dependencies for make as output 99
dependencies, make 60
DEPENDENCIES_OUTPUT 99
destructor function attribute 160
destructors vs goto 191
detecting '-traditional' 34
dialect options 31
digits in constraint 176
directory options 64
divide instruction, 88k 78
dollar signs in identifier names 162
double-word arithmetic 149
downward funargs 143
DW bit (29k) 73

E

'E' in constraint 175
enumeration clash warnings 44
environment variables 98
error messages 230
escape sequences, traditional 33
exclamation point 177
exit 32
exit status and VMS 248
explicit register variables 184
expressions containing statements . . 141
expressions, compound, as lvalues . . . 148
expressions, conditional, as lvalues . . 148
expressions, constructor 154
extended asm 170
extensible constraints 177
extensions, ?: 148, 149
extensions, C language 141
extensions, C++ language 189
external declaration scope 219

F

'F' in constraint 176
fabs 32
fatal signal 233
ffs 32
file name suffix 28
file names 61
float as function value type 220
format function attribute 159
forwarding calls 146
fscanf, and constant strings 218
function attributes 157
function pointers, arithmetic 154
function prototype declarations 160
function, size of pointer to 154
functions in arbitrary sections 157
functions that have no side effects . . . 157
functions that never return 157
functions with printf or scanf style arguments 157

G

'g' in constraint 176
'G' in constraint 176
g++ 30
G++ 21
g++ 1.xx 31
g++ older version 31
g++, separate compiler 31
GCC 21
GCC_EXEC_PREFIX 98
generalized lvalues 148
genflags, crash on Sun 4 207
global offset table 95
global register after longjmp 186
global register variables 185
GLOBALDEF 246
GLOBALREF 246
GLOBALVALUEDEF 246
GLOBALVALUEREF 246
GNU CC command options 23
goto in C++ 191
goto with computed label 143
gp-relative references (MIPS) 87
gprof 49
grouping options 23

H

'H' in constraint 176
hardware models and configurations, specifying 66
header files and VMS 245
HPPA Options 88

I

'i' in constraint 175
'I' in constraint 175
i386 Options 87
IBM RS/6000 and PowerPC Options .. 79
IBM RT options 83
IBM RT PC 216
identifier names, dollar signs in 162
identifiers, names in assembler code 184
identifying source, compiler (88k) 75
implicit argument: return value 189
implied #pragma implementation 192
include files and VMS 245
incompatibilities of GNU CC 218
increment operators 233
initializations in expressions 154
initializers with labeled elements 155
initializers, non-constant 154
inline automatic for C++ member fns 169
inline functions 169
inline functions, omission of 169
inlining and C++ pragmas 193
installation trouble 205
installing GNU CC 103
installing GNU CC on the Sun 133
installing GNU CC on VMS 133
integrating function code 169
Intel 386 Options 87
interface and implementation headers, C++ 191
intermediate C version, nonexistent .. 21
interrupts, functions compiled for ... 161
invalid assembly code 233
invalid input 234
invoking g++ 31

K

kernel and user registers (29k) 73
keywords, alternate 187
known causes of trouble 205

L

labeled elements in initializers 155
labels as values 143
labs 32
language dialect options 31
large bit shifts (88k) 78
length-zero arrays 151
Libraries 62
LIBRARY_PATH 99
link options 61
load address instruction 176
local labels 142
local variables in macros 147
local variables, specifying registers .. 186
long long data types 149
longjmp 186
longjmp and automatic variables 33
longjmp incompatibilities 218
longjmp warnings 41
lvalues, generalized 148

M

'm' in constraint 174
M680x0 options 67
M88k options 75
machine dependent options 66
machine specific constraints 178
macro with variable arguments 152
macros containing asm 173
macros, inline alternative 169
macros, local labels 142
macros, local variables in 147
macros, statements in expressions ... 141
macros, types of arguments 147
main and the exit status 248
make 60
matching constraint 176
maximum operator 191
member fns, automatically inline .. 169
memcmp 32
memcpy 32
memory model (29k) 73
memory references in constraints 174
messages, warning 40
messages, warning and error 230
middle-operands, omitted 149
minimum operator 191
MIPS options 83
misunderstandings in C++ 224
mktemp, and constant strings 218
mode attribute 164

modifiers in constraints 177
multiple alternative constraints 177
multiprecision arithmetic 149

N

'n' in constraint 175
name augmentation 248
named return value in C++ 189
names used in assembler code 184
naming convention, implementation headers 192
naming types 146
nested functions 143
newline vs string constants 34
nocommon attribute 165
non-constant initializers 154
non-static inline function 169
noreturn function attribute 157

O

'o' in constraint 174
OBJC_INCLUDE_PATH.................. 99
Objective C 21
obstack_free....................... 120
OCS (88k)............................ 75
offsettable address 174
old-style function definitions......... 160
omitted middle-operands 149
open coding 169
operand constraints, asm 174
optimize options 53
options to control warnings 40
options, C++ 35
options, code generation 94
options, debugging 48
options, dialect........................ 31
options, directory search 64
options, GNU CC command 23
options, grouping 23
options, linking 61
options, optimization.................. 53
options, order 23
options, preprocessor.................. 58
order of evaluation, side effects 230
order of options 23
other directory, compilation in 127
output file option...................... 30
overloaded virtual fn, warning 47

P

'p' in constraint 176
packed attribute 165
parameter forward declaration 152
parser generator, Bison 107
PIC 95
pointer arguments 158
portions of temporary objects, pointers to 224
pragma, reason for not using 160
pragmas in C++, effect on inlining ... 193
pragmas, interface and implementation 192
preprocessing numbers 221
preprocessing tokens 221
preprocessor options 58
processor selection (29k) 73
prof 49
promotion of formal parameters 160
push address instruction 176

Q

'Q', in constraint 177
qsort, and global register variables 185
question mark 177

R

'r' in constraint 175
r0-relative references (88k) 76
ranges in case statements 156
read-only strings 218
register positions in frame (88k) 76
register variable after longjmp 186
registers 170
registers for local variables 186
registers in constraints 175
registers, global allocation 184
registers, global variables in 185
reordering, warning................... 44
reporting bugs 233
rest argument (in macro) 152
return value of main................. 248
return value, named, in C++ 189
return, in C++ function header 189
RS/6000 and PowerPC Options 79

RT options . . . 83
RT PC . . . 216
run-time options . . . 94

S

's' in constraint . . . 176
scanf, and constant strings . . . 218
scope of a variable length array . . . 151
scope of declaration . . . 222
scope of external declarations . . . 219
search path . . . 64
second include path . . . 59
section function attribute . . . 159
section variable attribute . . . 165
separate directory, compilation in . . . 127
sequential consistency on 88k . . . 76
setjmp . . . 186
setjmp incompatibilities . . . 218
shared strings . . . 218
shared VMS run time system . . . 248
side effect in ?: . . . 149
side effects, macro argument . . . 141
side effects, order of evaluation . . . 230
signature . . . 196
signature in C++, advantages . . . 197
signature member function default implementation . . . 197
signatures, C++ . . . 196
simple constraints . . . 174
sin . . . 32
sizeof . . . 147
smaller data references (88k) . . . 76
smaller data references (MIPS) . . . 87
SPARC options . . . 69
specified registers . . . 184
specifying compiler version and target machine . . . 65
specifying hardware config . . . 66
specifying machine version . . . 65
specifying registers for local variables . . . 186
sqrt . . . 32
sscanf, and constant strings . . . 218
stack checks (29k) . . . 74
stage1 . . . 108
start files . . . 129
statements inside expressions . . . 141
static data in C++, declaring and defining . . . 224
'stdarg.h' and RT PC . . . 83
storem bug (29k) . . . 74
strcmp . . . 32
strcpy . . . 32
string constants . . . 218
string constants vs newline . . . 34
strlen . . . 32
structure passing (88k) . . . 78
structures . . . 220
structures, constructor expression . . . 154
submodel options . . . 66
subscripting . . . 153
subscripting and function values . . . 153
subtype polymorphism, C++ . . . 196
suffixes for C++ source . . . 30
Sun installation . . . 133
suppressing warnings . . . 40
surprises in C++ . . . 224
SVr4 . . . 77
syntax checking . . . 40
synthesized methods, warning . . . 47

T

target machine, specifying . . . 65
target options . . . 65
tcov . . . 50
template debugging . . . 44
template instantiation . . . 193
temporaries, lifetime of . . . 224
thunks . . . 143
TMPDIR . . . 98
traditional C language . . . 32
type abstraction, C++ . . . 196
type alignment . . . 162
type attributes . . . 166
typedef names as function parameters . . . 219
typeof . . . 147

U

Ultrix calling convention . . . 216
undefined behavior . . . 233
undefined function value . . . 233
underscores in variables in macros . . . 147
underscores, avoiding (88k) . . . 75

union, casting to a 157
unions 220
unresolved references and -nostdlib 62

V

'V' in constraint 175
value after longjmp 186
'varargs.h' and RT PC 83
variable alignment 162
variable attributes 163
variable number of arguments 152
variable-length array scope 151
variable-length arrays 151
variables in specified registers 184
variables, local, in macros 147
Vax calling convention 216
VAX options 68
'VAXCRTL' 248
VMS and case sensitivity 248
VMS and include files 245
VMS installation 133
void pointers, arithmetic 154
void, size of pointer to 154
volatile applied to function 157

W

warning for enumeration conversions 44
warning for overloaded virtual fn 47
warning for reordering of member initializers 44
warning for synthesized methods 47
warning messages 40
warnings vs errors 230
whitespace 219

X

'X' in constraint 176

Z

zero division on 88k 77
zero-length arrays 151

Debugging with GDB

The GNU Source-Level Debugger

Edition 4.12, for GDB version
January 1994

Richard M. Stallman and Roland H. Pesch

(Send bugs and comments on GDB to bug-gdb@prep.ai.mit.edu.)
Debugging with GDB
TEXinfo 2.122-95q3 (Cygnus)
doc@cygnus.com

Copyright © 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995 Free Software Foundation, Inc.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA
Printed copies are available for $20 each.
ISBN 1-882114-11-6

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

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

Table of Contents

Summary of GDB **1**
Free software 1
Contributors to GDB 2

1 A Sample GDB Session **5**

2 Getting In and Out of GDB **9**
2.1 Invoking GDB 9
2.1.1 Choosing files 10
2.1.2 Choosing modes 11
2.2 Quitting GDB 12
2.3 Shell commands 13

3 GDB Commands **15**
3.1 Command syntax 15
3.2 Command completion 16
3.3 Getting help 17

4 Running Programs Under GDB **21**
4.1 Compiling for debugging 21
4.2 Starting your program 23
4.3 Your program's arguments 24
4.4 Your program's environment 24
4.5 Your program's working directory 26
4.6 Your program's input and output 26
4.7 Debugging an already-running process 27
4.8 Killing the child process 28
4.9 Additional process information 28
4.10 Debugging programs with multiple threads 29
4.11 Debugging programs with multiple processes 31

5 Stopping and Continuing **33**
5.1 Breakpoints, watchpoints, and exceptions 33
5.1.1 Setting breakpoints 34
5.1.2 Setting watchpoints 38
5.1.3 Breakpoints and exceptions 39
5.1.4 Deleting breakpoints 40
5.1.5 Disabling breakpoints 40
5.1.6 Break conditions 42
5.1.7 Breakpoint command lists 43

5.1.8 Breakpoint menus 45
5.2 Continuing and stepping 45
5.3 Signals 49
5.4 Stopping and starting multi-thread programs 50

6 Examining the Stack 53
6.1 Stack frames 53
6.2 Backtraces 54
6.3 Selecting a frame 55
6.4 Information about a frame 56
6.5 MIPS machines and the function stack 58

7 Examining Source Files 59
7.1 Printing source lines 59
7.2 Searching source files 61
7.3 Specifying source directories 61
7.4 Source and machine code 62

8 Examining Data 65
8.1 Expressions 65
8.2 Program variables 66
8.3 Artificial arrays 67
8.4 Output formats 68
8.5 Examining memory 69
8.6 Automatic display 71
8.7 Print settings 73
8.8 Value history 78
8.9 Convenience variables 79
8.10 Registers 81
8.11 Floating point hardware 83

9 Using GDB with Different Languages 85
9.1 Switching between source languages 85
9.1.1 List of filename extensions and languages 85
9.1.2 Setting the working language 86
9.1.3 Having GDB infer the source language 86
9.2 Displaying the language 87
9.3 Type and range checking 87
9.3.1 An overview of type checking 88
9.3.2 An overview of range checking 89
9.4 Supported languages 90
9.4.1 C and C++ 90
9.4.1.1 C and C++ operators 91
9.4.1.2 C and C++ constants 93

9.4.1.3 C++ expressions 93
9.4.1.4 C and C++ defaults 94
9.4.1.5 C and C++ type and range checks 95
9.4.1.6 GDB and C 95
9.4.1.7 GDB features for C++ 95
9.4.2 Modula-2 96
9.4.2.1 Operators 97
9.4.2.2 Built-in functions and procedures 98
9.4.2.3 Constants 100
9.4.2.4 Modula-2 defaults 100
9.4.2.5 Deviations from standard Modula-2 100
9.4.2.6 Modula-2 type and range checks 101
9.4.2.7 The scope operators :: and 101
9.4.2.8 GDB and Modula-2 102

10 Examining the Symbol Table 103

11 Altering Execution 107

11.1 Assignment to variables 107
11.2 Continuing at a different address 108
11.3 Giving your program a signal 109
11.4 Returning from a function 109
11.5 Calling program functions 110
11.6 Patching programs 110

12 GDB Files 111

12.1 Commands to specify files 111
12.2 Errors reading symbol files 115

13 Specifying a Debugging Target 119

13.1 Active targets 119
13.2 Commands for managing targets 119
13.3 Choosing target byte order 122
13.4 Remote debugging 123
13.4.1 The GDB remote serial protocol 123
13.4.1.1 What the stub can do for you 124
13.4.1.2 What you must do for the stub 125
13.4.1.3 Putting it all together 127
13.4.1.4 Communication protocol 128
13.4.1.5 Using the `gdbserver` program 129
13.4.1.6 Using the `gdbserve.nlm` program .. 131
13.4.2 GDB with a remote i960 (Nindy) 132

13.4.2.1 Startup with Nindy 132
13.4.2.2 Options for Nindy 133
13.4.2.3 Nindy reset command 133
13.4.3 The UDI protocol for AMD29K 133
13.4.4 The EBMON protocol for AMD29K 134
13.4.4.1 Communications setup 134
13.4.4.2 EB29K cross-debugging 136
13.4.4.3 Remote log 136
13.4.5 GDB with a Tandem ST2000 137
13.4.6 GDB and VxWorks 137
13.4.6.1 Connecting to VxWorks 138
13.4.6.2 VxWorks download 138
13.4.6.3 Running tasks 139
13.4.7 GDB and Hitachi microprocessors 139
13.4.7.1 Connecting to Hitachi boards 139
13.4.7.2 Using the E7000 in-circuit emulator 140
13.4.7.3 Special GDB commands for Hitachi micros 140
13.4.8 GDB and remote MIPS boards 141
13.4.9 Simulated CPU target 142

14 Controlling GDB 145

14.1 Prompt 145
14.2 Command editing 145
14.3 Command history 146
14.4 Screen size 147
14.5 Numbers 148
14.6 Optional warnings and messages 148

15 Canned Sequences of Commands 151

15.1 User-defined commands 151
15.2 User-defined command hooks 152
15.3 Command files 153
15.4 Commands for controlled output 154

16 Using GDB under GNU Emacs 157

17 Reporting Bugs in GDB 161

17.1 Have you found a bug? 161
17.2 How to report bugs 161

Appendix A Command Line Editing 165
A.1 Introduction to Line Editing 165
A.2 Readline Interaction 165
A.2.1 Readline Bare Essentials 165
A.2.2 Readline Movement Commands 166
A.2.3 Readline Killing Commands 167
A.2.4 Readline Arguments 167
A.3 Readline Init File 168
A.3.1 Readline Init Syntax 168
A.3.1.1 Commands For Moving 170
A.3.1.2 Commands For Manipulating The History 170
A.3.1.3 Commands For Changing Text 171
A.3.1.4 Killing And Yanking 172
A.3.1.5 Specifying Numeric Arguments 172
A.3.1.6 Letting Readline Type For You 173
A.3.1.7 Some Miscellaneous Commands 173
A.3.2 Readline Vi Mode 173

Appendix B Using History Interactively 175
B.1 History Interaction 175
B.1.1 Event Designators 175
B.1.2 Word Designators 175
B.1.3 Modifiers 176

Appendix C Formatting Documentation 177

Appendix D Installing GDB 179
D.1 Compiling GDB in another directory 180
D.2 Specifying names for hosts and targets 181
D.3 `configure` options 182

Index 185

Summary of GDB

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

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

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

You can use GDB to debug programs written in C or C++. For more information, see Section 9.4.1 “C and C++,” page 90.

Support for Modula-2 and Chill is partial. For information on Modula-2, see Section 9.4.2 “Modula-2,” page 96. There is no further documentation on Chill yet.

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

GDB can be used to debug programs written in Fortran, although it does not yet support entering expressions, printing values, or similar features using Fortran syntax. It may be necessary to refer to some variables with a trailing underscore.

Free software

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

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

Contributors to GDB

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

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

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

So that they may not regard their 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 CX-UX support. Chris Smith contributed Convex support (and Fortran debugging). Jonathan Stone contributed Pyramid support. Michael Tiemann

contributed SPARC support. Tim Tucker contributed support for the Gould NP1 and Gould Powernode. Pace Willison contributed Intel 386 support. Jay Vosburgh contributed Symmetry support.

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 this manual.

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

Hitachi America, Ltd. sponsored the support for 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.

1 A Sample GDB Session

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

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

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

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

foo
0000
define(bar,defn(‘foo’))

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

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

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

```
$ gdb m4
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
 for details.

GDB , Copyright 1995 Free Software Foundation, Inc...
(gdb)
```

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

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

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

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

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

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

foo
0000
```

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

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

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

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

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

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

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

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

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

```
#4  0x79dc in expand_input () at macro.c:40
#5  0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

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

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

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

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

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

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

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

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

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

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

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

```
(gdb) c
Continuing.

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

baz
0000
```

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

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

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

```
(gdb) quit
```

2 Getting In and Out of GDB

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

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

2.1 Invoking GDB

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

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

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

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

```
gdb program
```

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

```
gdb program core
```

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

```
gdb program 1234
```

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

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

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

```
gdb -silent
```

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

Type

```
gdb -help
```

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

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

2.1.1 Choosing files

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

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

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

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

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

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

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

`-command` *`file`*
`-x` *`file`*
: Execute GDB commands from file *`file`*. See Section 15.3 “Command files,” page 153.

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

`-m`
`-mapped` *Warning:* this option depends on operating system facilities that are not supported on all systems.

If memory-mapped files are available on your system through the `mmap` system call, you can use this option to have GDB write the symbols from your program into a reusable file in the current directory. If the program you are debugging is called '`/tmp/fred`', the mapped symbol file is '`./fred.syms`'. Future GDB debugging sessions notice the presence of this file, and can quickly map in symbol information from it, rather than reading the symbol table from the executable program.

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

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

The `-mapped` and `-readnow` options are typically combined in order to build a '`.syms`' file that contains complete symbol information. (See Section 12.1 "Commands to specify files," page 111, for information

a '`.syms`' file for future use is:

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

2.1.2 Choosing modes

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

`-nx`
`-n` Do not execute commands 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 Section 15.3 "Command files," page 153.

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

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

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

```
Program exited normally.
```

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

`-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-to-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.

2.2 Quitting GDB

`quit` To exit GDB, use the `quit` command (abbreviated `q`), or type an end-of-file character (usually `C-d`).

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

If you have been using GDB to control an attached process or device, you can release it with the `detach` command (see Section 4.7 "Debugging an already-running process," page 27).

2.3 Shell commands

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

`shell` *command string*
: Invoke a the standard shell to execute *command string*. If it exists, the environment variable `SHELL` determines which shell to run. Otherwise GDB uses `/bin/sh`.

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

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

3 GDB Commands

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

3.1 Command syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in '`step 5`'. You can also use the `step` command with no arguments. Some command names do not allow any arguments.

GDB command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. You can test abbreviations by using them as arguments to the `help` command.

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

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

GDB can also use RET in another way: to partition lengthy output, in a way similar to the common utility `more` (see Section 14.4 "Screen size," page 147). Since it is easy to press one RET too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files (see Section 15.3 "Command files," page 153).

3.2 Command completion

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

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

```
(gdb) info bre TAB
```

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

```
(gdb) info breakpoints
```

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

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

```
(gdb) b make_ TAB
```

GDB sounds bell; press TAB again, to see:

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

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

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

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

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

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

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

```
(gdb) b bub TAB
```

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

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

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

3.3 Getting help

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

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

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

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, here is the help display for the class `status`:

```
(gdb) help status
Status inquiries.

List of commands:

show -- Generic command for showing things set
 with "set"
info -- Generic command for printing status

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

`help` *command*

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

`complete` *args*

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

```
complete i
```

results in:

```
info
inspect
ignore
```

This is intended for use by GNU Emacs.

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

under `show` in the Index point to all the sub-commands. See "Index," page 185.

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

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

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

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

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

`show version`
: Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may 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.

4 Running Programs Under GDB

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

4.1 Compiling for debugging

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

To request debugging information, specify the '`-g`' option when you run the compiler.

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

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.

4.2 Starting your program

`run`
`r`
: Use the `run` command to start your program under GDB. You must first specify the program name (except on VxWorks) with an argument to GDB (see Chapter 2 "Getting In and Out of GDB," page 9), or by using the `file` or `exec-file` command (see Section 12.1 "Commands to specify files," page 111).

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

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

The *arguments.*
: Specify the arguments to give your program as the arguments of the `run` command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the `SHELL` environment variable. See Section 4.3 "Your program's arguments," page 24.

The *environment.*
: Your program normally inherits its environment from GDB, but you can use the GDB commands `set environment` and `unset environment` to change parts of the environment that affect your program. See Section 4.4 "Your program's environment," page 24.

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

The *standard input and output.*
: Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the `run` command line, or you can use

the `tty` command to set a different device for your program. See Section 4.6 "Your program's input and output," page 26.

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

When you issue the `run` command, your program begins to execute immediately. See Chapter 5 "Stopping and continuing," page 33, for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. See Chapter 8 "Examining Data," page 65.

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

4.3 Your program's arguments

The arguments to your program can be specified by the arguments of the `run` command. 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.

4.4 Your program's environment

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

`path directory`
: Add *directory* to the front of the PATH environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by ':' or 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, this command:

```
set env USER = foo
```

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

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

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'.

4.5 Your program's working directory

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

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See Section 12.1 "Commands to specify files," page 111.

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

`pwd`
: Print the GDB working directory.

4.6 Your program's input and output

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

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

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

```
run > outfile
```

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

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

```
tty /dev/ttyb
```

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

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

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

4.7 Debugging an already-running process

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

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

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

When using `attach`, you should first use the `file` command to specify the program running in the process and load its symbol table. See Section 12.1 "Commands to Specify Files," page 111.

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 press `RET` again after executing the command.

If you exit GDB or use the `run` command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see Section 14.6 “Optional warnings and messages,” page 148).

4.8 Killing the child process

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

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

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

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

4.9 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.

4.10 Debugging programs with multiple threads

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

GDB provides these facilities for debugging multi-thread programs:

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

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

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

current thread. Debugging commands show program information from the perspective of the current thread.

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

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

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

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

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

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

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

For example,

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

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

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

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

`thread apply` [*threadno*] [*all*] *args*
: The `thread apply` command allows you to apply a command to one or more threads. Specify the numbers of the

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

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

See Section 5.4 "Stopping and starting multi-thread programs," page 50, for more information about how GDB behaves when you stop and start programs with multiple threads.

See Section 5.1.2 "Setting watchpoints," page 38, for information about watchpoints in programs with multiple threads.

4.11 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 Section 4.7 "Attach," page 27). From that point on you can debug the child process just like any other process which you attached to.

5 Stopping and Continuing

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

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

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

5.1 Breakpoints, watchpoints, and 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 Section 5.1.1 "Setting breakpoints," page 34), 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 Section 5.1.3 "Breakpoints and exceptions," page 39).

In SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can now 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 Section 5.1.2 "Setting watchpoints," page 38), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See Section 8.6 "Automatic display," page 71.

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.

5.1.1 Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable '`$bpnum`' records the number of the breakpoints you've set most recently; see Section 8.9 "Convenience variables," page 79, for a discussion of what you can do with convenience variables.

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

`break` *`function`*
: Set a breakpoint at entry to function *`function`*. When using source languages that permit overloading of symbols, such as C++, *`function`* may refer to more than one possible place to break. See Section 5.1.8 "Breakpoint menus," page 45, 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 at line *`linenum`* in the current source file. 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 Chapter 6 "Examining the Stack," page 53). In

any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a `finish` command in the frame inside the selected frame—except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops.

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

`break . . . if` *`cond`*

Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. '. . .' stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.1.6 "Break conditions," page 42, for more information on breakpoint conditions.

`tbreak` *`args`*

Set a breakpoint enabled only for one stop. *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See Section 5.1.5 "Disabling breakpoints," page 40.

`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.

`hbreak` is mostly useful in EPROM/ROM code debugging, because it allows you to set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClite DSU, in which DSU generates traps when a program accesses some date or instruction address that is assigned to the debug registers.

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 Section 5.1.6 "Break conditions," page 42.

`thbreak` *args*

Set a hardware-assisted breakpoint enabled only for one stop. *args* are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support, and some target hardware may not have this support. See Section 5.1.5 "Disabling breakpoints," page 40. Also See Section 5.1.6 "Break conditions," page 42.

`rbreak` *regex*

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

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

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

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

Breakpoint Numbers

Type Breakpoint or watchpoint.

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

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

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

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

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

`info break` with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see Section 8.5 "Examining memory," page 69).

`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 Section 5.1.6 "Break conditions," page 42).

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

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

`maint info breakpoints`

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

`breakpoint`
: Normal, explicitly set breakpoint.

`watchpoint`
: Normal, explicitly set watchpoint.

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

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

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

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

5.1.2 Setting watchpoints

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

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 the value in *`expr`* changes. `watch` can be used with the new trap-generation provided by SPARClite DSU, in which DSU generates traps when a program accesses some date or instruction address that is assigned to the debug registers. For the data addresses, DSU facilitates the `watch` command.

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 `awatch` and the other with `rwatch`. GDB will reject the command if you try to mix watchpoints, so you must delete or disable unused watchpoint commands before setting new ones.

`rwatch` *`expr`*

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

`awatch` *`expr`*

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

`info watchpoints`

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

> *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.

5.1.3 Breakpoints and exceptions

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

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

You can use `info catch` to list active exception handlers. See Section 6.4 "Information about a frame," page 56.

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 Section 5.1 "Breakpoints; watchpoints; and exceptions," page 33).

With a conditional breakpoint (see Section 5.1.6 "Break conditions," page 42) that depends on the value of *`id`*, you can stop your program

when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

5.1.4 Deleting breakpoints

It is often necessary to eliminate a breakpoint 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 Section 6.3 “Selecting a frame,” page 55). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

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

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

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

5.1.5 Disabling breakpoints

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

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

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

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

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

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

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

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

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

Except for a breakpoint set with `tbreak` (see Section 5.1.1 “Setting breakpoints,” page 34), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see Section 5.2 “Continuing and stepping,” page 45.)

5.1.6 Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language (see Section 8.1 “Expressions,” page 65). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

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

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

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see Section 5.1.7 “Breakpoint command lists,” page 43).

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the `break` command. See Section 5.1.1 “Setting breakpoints,” page 34. 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 evalu-

ate *expression* at the time the `condition` command is given, however. See Section 8.1 “Expressions,” page 65.

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

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

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

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

  When you use `continue` to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to `continue`, rather than using `ignore`. See Section 5.2 “Continuing and stepping,” page 45.

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

  You could achieve the effect of the ignore count with a condition such as ‘`$foo-- <= 0`’ using a debugger convenience variable that is decremented each time. See Section 8.9 “Convenience variables,” page 79.

5.1.7 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).

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

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

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

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

The commands `echo`, `output`, and `printf` allow you to print precisely controlled output, and are often useful in silent breakpoints. See Section 15.4 “Commands for controlled output,” page 154.

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

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

One application for breakpoint commands is to compensate for one bug so you can test for another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the `continue` command so

that your program does not stop, and start with the `silent` command so that no output is produced. Here is an example:

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

5.1.8 Breakpoint menus

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

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

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

5.2 Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one

more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If due to a signal, you may want to use `handle`, or use '`signal 0`' to resume execution. See Section 5.3 "Signals," page 49.)

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

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore` (see Section 5.1.6 "Break conditions," page 42).

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 Section 11.4 "Returning from a function," page 109) to go back to the calling function; or `jump` (see Section 11.2 "Continuing at a different address," page 108) to go to an arbitrary location in your program.

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

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

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

The `step` command 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 -gl` on MIPS machines. Previously, `step` entered subroutines if there was any debugging information about the routine.

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

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

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

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

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

Contrast this with the `return` command (see Section 11.4 “Returning from a function,” page 109).

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

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

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

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

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

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

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

Continue running your program until either the specified location is reached, or the current stack frame returns. *`location`* is any of the forms of argument acceptable to `break` (see Section 5.1.1 “Setting breakpoints,” page 34). This form of the command uses breakpoints, and hence is quicker than `until` without an argument.

`stepi`
`si`

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

It is often useful to do ‘`display/i $pc`’ when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See Section 8.6 “Automatic display,” page 71.

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

`nexti`
`ni`

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

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

5.3 Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in Unix `SIGINT` is the signal a program gets when you type an interrupt (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 are:

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

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

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

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

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

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

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

You can also use the `signal` command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with '`signal 0`'. See Section 11.3 "Giving your program a signal," page 109.

5.4 Stopping and starting multi-thread programs

When your program has multiple threads (see Section 4.10 "Debugging programs with multiple threads," page 29), you can choose whether to set breakpoints on all threads, or on a particular thread.

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

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

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

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

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

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

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

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

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

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

6 Examining the Stack

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

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

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in. See Section 6.3 “Selecting a frame,” page 55.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command (see Section 6.4 “Information about a frame,” page 56).

6.1 Stack frames

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

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

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

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

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

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

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

6.2 Backtraces

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

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

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

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

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

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

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

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

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

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

6.3 Selecting a frame

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

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

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

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

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

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

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

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

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

For example:

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

After such a printout, the `list` command with no arguments prints ten lines centered on the point of execution in the frame. See Section 7.1 “Printing source lines,” page 59.

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

6.4 Information about a frame

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

`frame`
`f`
: When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated `f`. With an argument, this command is used to select a stack frame. See Section 6.3 “Selecting a frame,” page 55.

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

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

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

`info frame` *`addr`*
`info f` *`addr`*
Print a verbose description of the frame at address *`addr`*, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the `frame` command. See Section 6.3 “Selecting a frame,” page 55.

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

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

`info catch`
Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the `up`, `down`, or `frame` commands); then type `info catch`. See Section 5.1.3 “Breakpoints and exceptions,” page 39.

6.5 MIPS machines and the function stack

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

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

`set heuristic-fence-post` *`limit`*
: Restrict GDB to examining at most *`limit`* bytes in its search for the beginning of a function. A value of *`0`* (the default) means there is no limit. However, except for *`0`*, the larger the limit the more bytes `heuristic-fence-post` must search and therefore the longer it takes to run.

`show heuristic-fence-post`
: Display the current limit.

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

7 Examining Source Files

GDB can print parts of your program's source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see Section 6.3 "Selecting a frame," page 55), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source; see Chapter 16 "Using GDB under GNU Emacs," page 157.

7.1 Printing source lines

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

Here are the forms of the `list` command most commonly used:

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

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

`list`
: Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see Chapter 6 "Examining the Stack," page 53), this prints lines centered around that line.

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

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

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

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

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

In general, the `list` command expects you to supply zero, one or two *linespecs*. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. Here is a complete description of the possible arguments for `list`:

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

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

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

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

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

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

`list`
: As described in the preceding table.

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

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

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

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

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

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

`filename``:`*`function`*
: Specifies the line of the open-brace that begins the body of the function *`function`* in the file *`filename`*. You only need the file

name with a function name to avoid ambiguity when there are identically named functions in different source files.

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

7.2 Searching source files

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

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

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

7.3 Specifying source directories

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

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

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

When you start GDB, its source path 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 as follows:

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

7.4 Source and machine code

You can use the command `info line` to map source lines to program addresses (and vice versa), and the command `disassemble` to display a range of addresses as machine instructions. When run under GNU Emacs mode, the `info line` command 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*. You can specify source lines in any of the ways understood by the `list` command (see Section 7.1 "Printing source lines," page 59).

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

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

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

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

After `info line`, the default address for the `x` command is changed to the starting address of the line, so that 'x/i' is sufficient to begin examining the machine code (see Section 8.5 "Examining memory," page 69). Also, this address is saved as the value of the convenience variable `$_` (see Section 8.9 "Convenience variables," page 79).

`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>:     b 0x63fc <builtin_init+5364>
0x63f4 <builtin_init+5356>:     ld [%o0+4], %o0
0x63f8 <builtin_init+5360>:     or %o0, 0x1a4, %o0
0x63fc <builtin_init+5364>:     call 0x9288 <path_search>
0x6400 <builtin_init+5368>:     nop
End of assembler dump.
```

8 Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym `inspect`. It evaluates and prints the value of an expression of the language your program is written in (see Chapter 9 "Using GDB with Different Languages," page 85).

`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 Section 8.4 "Output formats," page 68.

`print`
`print /`*f*
: If you omit *exp*, GDB displays the last value again (from the *value history*; see Section 8.8 "Value history," page 78). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See Section 8.5 "Examining memory," page 69.

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 Chapter 10 "Examining the Symbol Table," page 103.

8.1 Expressions

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

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

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See Chapter 9 "Using GDB with Different Languages," page 85, for information on how to use expressions in other languages.

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

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

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

`@`
: '`@`' is a binary operator for treating parts of memory as arrays. See Section 8.3 "Artificial arrays," page 67, for more information.

`::`
: '`::`' allows you to specify a variable in terms of the file or function where it is defined. See Section 8.2 "Program variables," page 66.

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

8.2 Program variables

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

Variables in expressions are understood in the selected stack frame (see Section 6.3 "Selecting a frame," page 55); they must be either:

global (or static)

or

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

This means that in the function

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

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

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

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

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

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

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

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

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

8.3 Artificial arrays

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

You can do this by referring to a contiguous span of memory as an *artificial array*, using the binary operator '`@`'. The left operand of '`@`' should be the first element of the desired array and be an individual object. The right operand should be the desired length of the array. The

result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

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

you can print the contents of `array` with

```
p *array@len
```

The left operand of '`@`' must reside in memory. Array values made with '`@`' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see Section 8.8 "Value history," page 78), after printing one out.

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

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

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

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

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see Section 8.9 "Convenience variables," page 79) as a counter in an expression that prints the first interesting value, and then repeat that expression via RET. For instance, suppose you have an array `dtab` of pointers to structures, and you are interested in the values of a field `fv` in each structure. Here is an example of what you might type:

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

8.4 Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data

in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

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

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

`d` Print as integer in signed decimal.

`u` Print as integer in unsigned decimal.

`o` Print as integer in octal.

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

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

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

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

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

For example, to print the program counter in hex (see Section 8.10 "Registers," page 81), type

```
p/x $pc
```

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

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

8.5 Examining memory

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

[1] '`b`' cannot be used because these format letters are also used with the `x` command, where '`b`' stands for "byte"; see Section 8.5 "Examining memory," page 69.

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

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

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

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

u, the unit size
: The unit size is any of

  `b` Bytes.

  `h` Halfwords (two bytes).

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

  `g` Giant words (eight bytes).

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

addr, starting display address
: *addr* is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. See Section 8.1 "Expressions," page 65, for more information on expressions. The default for *addr* is usually just after the last address examined—but several other commands also set the default address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, '`x/3uh 0x54320`' is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers ('u'), starting at address `0x54320`. '`x/4xw $sp`' prints the four words ('w') of memory above

the stack pointer (here, '$sp'; see Section 8.10 "Registers," page 81) in hexadecimal ('x').

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

Even though the unit size *u* is ignored for the formats 's' and 'i', you might still want to use a count *n*; for example, '3i' specifies that you want to see three machine instructions, including any operands. The command disassemble gives an alternative way of inspecting machine instructions; see Section 7.4 "Source and machine code," page 62.

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

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

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

8.6 Automatic display

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

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

This display shows item numbers, expressions and their current values. As with displays you request manually using x or print, you can specify the output format you prefer; in fact, display decides whether to use

`print` or `x` depending on how elaborate your format specification is—it uses `x` if you specify a unit size, or one of the two formats ('`i`' and '`s`') that are only supported by `x`; otherwise it uses `print`.

`display` *exp*
: Add the expression *exp* to the list of expressions to display each time your program stops. See Section 8.1 "Expressions," page 65.

  `display` does not repeat if you press RET 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 Section 8.4 "Output formats," page 68.

`display/`*fmt addr*
: For *fmt* '`i`' or '`s`', or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing '`x/`*fmt addr*'. See Section 8.5 "Examining memory," page 69.

For example, '`display/i $pc`' can be helpful, to see the machine instruction about to be executed each time execution stops ('`$pc`' is a common name for the program counter; see Section 8.10 "Registers," page 81).

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

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

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

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

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

`info display`

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

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

8.7 Print settings

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

These settings are useful for debugging programs in any language:

`set print address`
`set print address on`

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

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

`set print address off`

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

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

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

`show print address`
: Show whether or not addresses are to be printed.

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

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

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

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

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

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

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

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

If you have a pointer and you are not sure where it points, try '`set print symbol-filename on`'. Then you can determine the name and source file location of the variable where it points, using '`p/a` *`pointer`*'.

This interprets the address in symbolic form. For example, here GDB shows that a variable `ptt` points at another variable `t`, defined in '`hi2.c`':

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

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

Other settings control how different kinds of objects are printed:

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

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

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

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

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

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

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

This is the default format.

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

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

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

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

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

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

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

For example, given the declarations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`foo`
: Show the list of formats.

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

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

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

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

`set print vtbl`
`set print vtbl on`
: Pretty print C++ virtual function tables. The default is off.

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

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

8.8 Value history

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

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

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

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

```
p *$
```

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

```
p *$.next
```

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

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

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

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

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

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

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

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

8.9 Convenience variables

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

Convenience variables are prefixed with '$'. Any name preceded by '$' can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see Section 8.10 "Registers," page 81). (Value history references, in contrast, are *numbers* preceded by '$'. See Section 8.8 "Value history," page 78.)

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 example, to print a field from successive elements of an array of structures:

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

Repeat that command by typing `RET`.

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

`$_`
: The variable `$_` is automatically set by the `x` command to the last address examined (see Section 8.5 "Examining memory," page 69). 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.

8.10 Registers

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

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

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

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

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

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer[2] with

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

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The `info registers` command shows the canonical names. For example, on the SPARC, `info`

---

[2] This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use `return`; see Section 11.4 "Returning from a function," page 109.

`registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`.

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

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

Normally, register values are relative to the selected stack frame (see Section 6.3 "Selecting a frame," page 55). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with '`frame 0`').

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

`set rstack_high_address` *`address`*
: On AMD 29000 family processors, registers are saved in a separate "register stack". There is no way for GDB to determine the extent of this stack. Normally, GDB just assumes that the stack is "large enough". This may result in GDB referencing memory locations that do not exist. If necessary, you can get around this problem by specifying the ending address of the register stack with the `set rstack_high_address` command. The argument should be an address, which you probably want to precede with '`0x`' to specify in hexadecimal.

`show rstack_high_address`
: Display the current limit of the register stack, on AMD 29000 family processors.

8.11 Floating point hardware

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

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

9 Using GDB with Different Languages

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

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

9.1 Switching between source languages

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

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

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

9.1.1 List of filename extensions and languages

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

'`.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.

9.1.2 Setting the working language

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

If you wish, you may set the language manually. To do this, issue the command '`set language` *`lang`*', where *`lang`* is the name of a language, such as `c` or `modula-2`. For a list of the supported languages, type '`set language`'.

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

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

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

9.1.3 Having GDB infer the source language

To have GDB set the working language automatically, use '`set language local`' or '`set language auto`'. GDB then infers the working language. That is, when your program stops in a frame (usually by encountering a breakpoint), GDB sets the working language to the language recorded for the function in that frame. If the language for a frame is unknown (that is, if the function or block corresponding to the

frame was defined in a source file that does not have a recognized extension), the current working language is not changed, and GDB issues a warning.

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

9.2 Displaying the language

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

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

`info frame`
: Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See Section 6.4 "Information about a frame," page 56, to identify the other information listed here.

`info source`
: Display the source language of this source file. See Chapter 10 "Examining the Symbol Table," page 103, to identify the other information listed here.

9.3 Type and range checking

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

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

GDB can check for conditions like the above if you wish. Although GDB does not check the statements in your program, it can check ex-

pressions entered directly into GDB for evaluation via the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your program's source language. See Section 9.4 “Supported languages,” page 90, for the default settings of supported languages.

9.3.1 An overview of type checking

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

```
1 + 2 ⇒ 3
```

but

```
error  1 + 2.3
```

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

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

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

Each language defines to what degree it is strict about type. For instance, both Modula-2 and C require the arguments to arithmetical operators to be numbers. In C, enumerated types and pointers can be represented as numbers, so that they are valid arguments to mathematical operators. See Section 9.4 “Supported languages,” page 90, for further details on specific languages.

GDB provides some additional commands for controlling the type checker:

`set check type auto`
: Set type checking on or off based on the current working language. See Section 9.4 “Supported languages,” page 90, 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.

9.3.2 An overview of range checking

In some languages (such as Modula-2), it is an error to exceed the bounds of a type; this is enforced with run-time checks. Such range checking is meant to ensure program correctness by making sure computations do not overflow, or indices on an array element access do not exceed the bounds of the array.

For expressions you use in GDB commands, you can tell GDB to treat range errors in one of three ways: ignore them, always treat them as errors and abandon the expression, or issue warnings but evaluate the expression anyway.

A range error can result from numerical overflow, from exceeding an array index bound, or when you type a constant that is not a member of any type. Some languages, however, do not treat overflows as an error. In many implementations of C, mathematical overflow causes the result to "wrap around" to lower values—for example, if *m* is the largest integer value, and *s* is the smallest, then

```
m + 1 ⇒ s
```

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. See Section 9.4 "Supported languages," page 90, for further details on specific languages.

GDB provides some additional commands for controlling the range checker:

`set check range auto`
: Set range checking on or off based on the current working language. See Section 9.4 “Supported languages,” page 90, 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.

9.4 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 Section 8.1 “Expressions,” page 65) can be used with the constructs of any supported language.

The following sections detail to what degree each source language is supported by GDB. These sections are not meant to be language tutorials or references, but serve only as a reference guide to what the GDB expression parser accepts, and what input and output formats should look like for different languages. There are many good books written on each of these languages; please look to these for a language reference or tutorial.

9.4.1 C and C++

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

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

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

9.4.1.1 C and C++ operators

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

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

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

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

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

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

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

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

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

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

`|`
: Bitwise OR. Defined on integral types.

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

`&`
: Bitwise AND. Defined on integral types.

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

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

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

`@`
: The GDB "artificial array" operator (see Section 8.1 "Expressions," page 65).

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

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

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

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

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

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

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

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

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

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

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

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

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

`::` Doubled colons also represent the GDB scope operator (see Section 8.1 “Expressions,” page 65). Same precedence as `::`, above.

9.4.1.2 C and C++ constants

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

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

9.4.1.3 C++ expressions

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

1. Member function calls are allowed; you can use expressions like
   ```
   count = aml->GetOriginal(x, y)
   ```
2. While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer `this` following the same rules as C++.
3. You can call overloaded functions; GDB resolves the function call to the right definition, with 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.
4. GDB understands variables declared as C++ references; you can use them in expressions just as you do in C++ source—they are automatically dereferenced.

   In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The *address* of a reference variable is always shown, unless you have specified '`set print address off`'.
5. GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in an expression like '`scope1::scope2::name`'. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see Section 8.2 "Program variables," page 66).

9.4.1.4 C and C++ defaults

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

This happens regardless of whether you or GDB selects the working language.

If you allow GDB to set the language automatically, it recognizes source files whose names end with '`.c`', '`.C`', or '`.cc`', and when GDB enters code compiled from one of these files, it sets the working language to C or C++. See Section 9.1.3 "Having GDB infer the source language," page 86, for further details.

9.4.1.5 C and C++ type and range checks

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

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

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

9.4.1.6 GDB and C

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

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See Section 8.1 "Expressions," page 65.

9.4.1.7 GDB features for C++

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

breakpoint menus
: When you want a breakpoint in a function whose name is overloaded, GDB breakpoint menus help you specify which function definition you want. See Section 5.1.8 "Breakpoint menus," page 45.

`rbreak` *regex*
: Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not

members of any special classes. See Section 5.1.1 "Setting breakpoints," page 34.

`catch` *exceptions*
`info catch`
: Debug C++ exception handling using these commands. See Section 5.1.3 "Breakpoints and exceptions," page 39.

`ptype` *typename*
: Print inheritance relationships as well as other information for type *typename*. See Chapter 10 "Examining the Symbol Table," page 103.

`set print demangle`
`show print demangle`
`set print asm-demangle`
`show print asm-demangle`
: Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. See Section 8.7 "Print settings," page 73.

`set print object`
`show print object`
: Choose whether to print derived (actual) or declared types of objects. See Section 8.7 "Print settings," page 73.

`set print vtbl`
`show print vtbl`
: Control the format for printing virtual function tables. See Section 8.7 "Print settings," page 73.

Overloaded symbol names
: You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type `symbol(types)` rather than just *symbol*. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See Section 3.2 "Command completion," page 16, for details on how to do this.

9.4.2 Modula-2

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

9.4.2.1 Operators

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

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

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

`,` Function argument or array index separator.

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

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

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

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

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

`OR` Boolean disjunction. Defined on boolean types.

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

`@` The GDB “artificial array” operator (see Section 8.1 “Expressions,” page 65).

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

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

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

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

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

`^` Pointer dereferencing. Defined on pointer types.

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

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

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

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

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

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

9.4.2.2 Built-in functions and procedures

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

a represents an `ARRAY` variable.

c represents a `CHAR` constant or variable.

i represents a variable or constant of integral type.

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

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

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

t represents a type.

v represents a variable.

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

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

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

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

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

`DEC(v)`
: Decrements the value in the variable *v*. 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(r)`
: Returns the integral part of *r*.

`VAL(t,i)`
: Returns the member of the type *t* whose ordinal value is *i*.

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

9.4.2.3 Constants

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

- Integer constants are simply a sequence of digits. When used in an expression, a constant is interpreted to be type-compatible with the rest of the expression. Hexadecimal integers are specified by a trailing 'H', and octal integers by a trailing 'B'.
- Floating point constants appear as a sequence of digits, followed by a decimal point and another sequence of digits. An optional exponent can then be specified, in the form 'E[+|-]*nnn*', where '[+|-]*nnn*' is the desired exponent. All of the digits of the floating point constant must be valid decimal (base 10) digits.
- Character constants consist of a single character enclosed by a pair of like quotes, either single (') or double ("). They may also be expressed by their ordinal value (their ASCII value, usually) followed by a 'C'.
- String constants consist of a sequence of characters enclosed by a pair of like quotes, either single (') or double ("). Escape sequences in the style of C are also allowed. See Section 9.4.1.2 "C and C++ constants," page 93, for a brief explanation of escape sequences.
- Enumerated constants consist of an enumerated identifier.
- Boolean constants consist of the identifiers `TRUE` and `FALSE`.
- Pointer constants consist of integral values only.
- Set constants are not yet supported.

9.4.2.4 Modula-2 defaults

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

If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with '`.mod`' sets the working language to Modula-2. See Section 9.1.3 "Having GDB set the language automatically," page 86, for further details.

9.4.2.5 Deviations from standard Modula-2

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

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

9.4.2.6 Modula-2 type and range checks

> *Warning:* in this release, GDB does not yet perform type or range checking.

GDB considers two Modula-2 variables type equivalent if:

- They are of types that have been declared equivalent via a `TYPE` *t1* `=` *t2* statement
- They have been declared on the same line. (Note: This is true of the GNU Modula-2 compiler, but it may not be true of other compilers.)

As long as type checking is enabled, any attempt to combine variables whose types are not equivalent is an error.

Range checking is done on all mathematical operations, assignment, array index bounds, and all built-in functions and procedures.

9.4.2.7 The scope operators `::` and `.`

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

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

where *scope* is the name of a module or a procedure, *module* the name of a module, and *id* is any declared identifier within your program, except another module.

Using the `::` operator makes GDB search the scope specified by *scope* for the identifier *id*. If it is not found in the specified scope, then GDB searches all scopes enclosing the one specified by *scope*.

Using the . operator makes GDB search the current scope for the identifier specified by *id* that was imported from the definition module specified by *module*. With this operator, it is an error if the identifier *id* was not imported from definition module *module*, or if *id* is not an identifier in *module*.

9.4.2.8 GDB and Modula-2

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

The `@` operator (see Section 8.1 "Expressions," page 65), 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}adrexp`' is still useful. (see Section 8.1 "Expressions," page 65)

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

10 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 Section 2.1.1 "Choosing files," page 10), or by one of the file-management commands (see Section 12.1 "Commands to specify files," page 111).

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see Section 8.2 "Program variables," page 66). File names are recorded in object files as debugging symbols, but GDB would ordinarily parse a typical file name, like '`foo.c`', as the three words '`foo`' '`.`' '`c`'. To allow GDB to recognize '`foo.c`' as a single symbol, enclose it in single quotes; for example,

```
p 'foo.c'::x
```

looks up the value of `x` in the scope of the file '`foo.c`'.

`info address` *symbol*
: Describe where the data for *symbol* is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.

  Note the contrast with '`print &`*symbol*', which does not work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

`whatis` *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 Section 8.1 "Expressions," page 65.

`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 example, for this variable declaration:

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

the two commands give this output:

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

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

`info types regexp`
`info types`
Print a brief description of all types whose 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 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 Section 12.1 "Commands to specify files," page 111, for a discussion of how GDB reads symbols (in the description of `symbol-file`).

11 Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

For example, you can store new values into variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function.

11.1 Assignment to variables

To alter the value of a variable, evaluate an assignment expression. See Section 8.1 “Expressions,” page 65. For example,

```
print x=4
```

stores the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4). See Chapter 9 “Using GDB with Different Languages,” page 85, for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the expression's value is not printed and is not put in the value history (see Section 8.8 “Value history,” page 78). The expression is evaluated only for its effects.

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

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

The invalid expression, of course, is '`=47`'. In order to actually set the program's variable `width`, use

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

GDB allows more implicit conversions in assignments than C; you can freely store an integer value into a pointer variable or vice versa,

and you can convert any structure to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the '`{...}`' construct to generate a value of specified type at a specified address (see Section 8.1 "Expressions," page 65). For example, `{int}0x83040` refers to memory location `0x83040` as an integer (which implies a certain size and representation in memory), and

```
set {int}0x83040 = 4
```

stores the value 4 into that memory location.

11.2 Continuing at a different address

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

`jump` *`linespec`*

Resume execution at line *`linespec`*. Execution stops again immediately if there is a breakpoint there. See Section 7.1 "Printing source lines," page 59, 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 Section 5.2 "Continuing and stepping," page 45.

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

11.3 Giving your program a signal

`signal` *`signal`*

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

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

`signal` does not repeat when you press `RET` a second time after executing the command.

Invoking the `signal` command is not the same as invoking the `kill` utility from the shell. Sending a signal with `kill` causes GDB to decide what to do with the signal depending on the signal handling tables (see Section 5.3 "Signals," page 49). The `signal` command passes the signal directly to your program.

11.4 Returning from a function

`return`
`return` *`expression`*

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

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

This pops the selected stack frame (see Section 6.3 "Selecting a frame," page 55), and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The `return` command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the `finish` command (see Section 5.2 “Continuing and stepping,” page 45) resumes execution until the selected stack frame returns naturally.

11.5 Calling program functions

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

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

A new user-controlled variable, *`call_scratch_address`*, specifies the location of a scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack does not work in systems that have separate instruction and data spaces.

11.6 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` command) after changing `set write`, for your new setting to take effect.

`show write`
: Display whether executable files and core files are opened for writing as well as reading.

12 GDB Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, you must also tell GDB the name of the core dump file.

12.1 Commands to specify files

You may want to specify executable and core dump file names. The usual way to do this is at start-up time, using the arguments to GDB's start-up commands (see Chapter 2 "Getting In and Out of GDB," page 9).

Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify a file you want to use. In these situations the GDB commands to specify new files are useful.

`file` *`filename`*
: Use *`filename`* as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the `run` command. If you do not specify a directory and the file is not found in the GDB working directory, GDB uses the environment variable `PATH` as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the `path` command.

  On systems with memory-mapped files, an auxiliary file '*`filename`*`.syms`' may hold symbol table information for *`filename`*. If so, GDB maps in the symbol table from '*`filename`*`.syms`', starting up more quickly. See the descriptions of the file options '`-mapped`' and '`-readnow`' (available on the command line, and with the commands `file`, `symbol-file`, or `add-symbol-file`, described below), for more information.

`file`
: `file` with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file` [*`filename`*]
: Specify that the program to be run (but not the symbol table) is found in *`filename`*. GDB searches the environment variable `PATH` if necessary to locate your program. Omitting *`filename`* means to discard information on the executable file.

`symbol-file [ filename ]`

Read symbol table information from file *filename*. PATH is searched when necessary. Use the `file` command to get both symbol table and program to run from the same file.

`symbol-file` with no argument clears out GDB information on your program's symbol table.

The `symbol-file` command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

`symbol-file` does not repeat if you press RET again after executing it once.

When GDB is configured for a particular environment, it understands debugging information in whatever format is the standard generated for that environment; you may use either a GNU compiler, or other compilers that adhere to the local conventions. 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 Section 14.6 “Optional warnings and messages,” page 148.)

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 that the core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the `kill` command (see Section 4.8 "Killing the child process," page 28).

`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 Section 13.4.7 "GDB and Hitachi Microprocessors," page 139), 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 press `RET` 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 `add-symbol-file` command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `symbol-file` command.

`add-symbol-file` does not repeat if you press `RET` after using it.

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

`add-shared-symbol-file`

The `add-shared-symbol-file` command can be used only under Harris' CXUX operating system for the Motorola 88k. GDB automatically looks for shared libraries, however if GDB does not find yours, you can run `add-shared-symbol-file`. It takes no arguments.

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

`info files`
`info target`
: `info files` and `info target` are synonymous; both print the current target (see Chapter 13 "Specifying a Debugging Target," page 119), including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command `help target` lists all possible targets rather than current ones.

All file-specifying commands allow both absolute and relative file names as arguments. GDB always converts the file name to an absolute file name and remembers it that way.

GDB supports SunOS, SVr4, Irix 5, and IBM RS/6000 shared libraries. GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library, however—unless you are debugging a core file).

`info share`
`info sharedlibrary`
: Print the names of the shared libraries which are currently loaded.

`sharedlibrary` *regex*
`share` *regex*
: Load shared object library symbols for files matching a Unix regular expression. As with files loaded automatically, it only loads shared libraries required by your program for a core file or after typing `run`. If *regex* is omitted all shared libraries required by your program are loaded.

12.2 Errors reading symbol files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed

symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the `set complaints` command (see Section 14.6 “Optional warnings and messages,” page 148).

The messages currently printed, and their meanings, include:

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

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

`block at` *address* `out of order`
: The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so.

  GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine what source file is affected by specifying `set verbose on`. See Section 14.6 “Optional warnings and messages,” page 148.)

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

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

`bad string table offset in symbol` *n*
: Symbol number *n* contains a pointer into the string table which is larger than the size of the string table.

  GDB circumvents the problem by considering the symbol to have the name `foo`, which may cause other problems if many symbols end up with this name.

`unknown symbol type 0x`*nn*
: The symbol information contains new data types that GDB does not yet know how to read. `0x`*nn* is the symbol type of the 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.

13 Specifying a Debugging Target

A *target* is the execution environment occupied by your program. Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the `file` or `core` commands. When you need more flexibility—for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection—you can use the `target` command to specify one of the target types configured for GDB (see Section 13.3 “Commands for managing targets,” page 122).

13.1 Active targets

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

For example, if you execute ‘`gdb a.out`’, then the executable file `a.out` is the only active target. If you designate a core file as well—presumably from a prior run that crashed and coredumped—then GDB has two active targets and uses them in tandem, looking first in the corefile target, then in the executable file, to satisfy requests for memory addresses. (Typically, these two classes of target are complementary, since core files contain only a program’s read-write memory—variables and so on—plus machine status, while executable files contain only the program text and initialized data.)

When you type `run`, your executable file becomes an active process target as well. When a process target is active, all GDB commands requesting memory addresses refer to that target; addresses in an active core file or executable file target are obscured while the process target is active.

Use the `core-file` and `exec-file` commands to select a new core file or executable target (see Section 12.1 “Commands to specify files,” page 111). To specify as a target a process that is already running, use the `attach` command (see Section 4.7 “Debugging an already-running process,” page 27).

13.2 Commands for managing targets

`target` *`type parameters`*
: Connects the GDB host environment to a target machine or process. A target is typically a protocol for talking to

debugging facilities. You use the argument *type* to specify the type or protocol of the target machine.

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

The `target` command does not repeat if you press `RET` again after executing the command.

`help target`
: Displays the names of all targets available. To display targets currently selected, use either `info target` or `info files` (see Section 12.1 “Commands to specify files,” page 111).

`help target` *name*
: Describe a particular target, including any parameters necessary to select it.

`set gnutarget` *args*
: GDB uses its own library BFD to read your files and knows whether it is reading an *executable*, a *core*, or a *.o* file. However you can specify the file format if you want with the `set gnutarget` command. Unlike most `target` commands, with `gnutarget` the target is a program, not a machine.

  Warning: To specify a file format with `set gnutarget`, you must know the actual BFD name.

  See Section 12.1 “Commands to specify files,” page 111.

`show gnutarget`
: Use the `show gnutarget` command to display what file format `gnutarget` is set to read. If you have not set `gnutarget`, GDB will determine the file format for each file automatically and `show gnutarget` displays: `The current BDF target is "auto"`.

Here are some common targets (available or not depending on the GDB configuration). Wherever it is not specified, *dev* is the serial device as for `target remote`:

`target exec` *program*
: An executable file. ‘`target exec` *program*’ is the same as ‘`exec-file` *program*’.

`target core` *filename*
: A core dump file. ‘`target core` *filename*’ is the same as ‘`core-file` *filename*’.

`target remote` *dev*
: Remote serial target in GDB-specific protocol. The argument *dev* specifies what serial device to use for the connection

(e.g. '/dev/ttya'). See Section 13.4 "Remote debugging," page 123. 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 Section 13.4.9 "Simulated CPU Target," page 142.

target udi *keyword*
: Remote AMD29K target, using the AMD UDI protocol. The *keyword* argument specifies which 29K board or simulator to use. See Section 13.4.3 "The UDI protocol for AMD29K," page 133.

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 Section 13.4.4 "The EBMON protocol for AMD29K," page 134.

target hms *dev*
: A Hitachi SH, H8/300, or H8/500 board, attached via serial line to your host. Use special commands device and speed to control the serial line and the communications speed used. See Section 13.4.7 "GDB and Hitachi Microprocessors," page 139.

target nindy *devicename*
: An Intel 960 board controlled by a Nindy Monitor. *devicename* is the name of the serial device to use for the connection, e.g. '/dev/ttya'. See Section 13.4.2 "GDB with a remote i960 (Nindy)," page 132.

target st2000 *dev speed*
: A Tandem ST2000 phone switch, running Tandem's STDBUG 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 Section 13.4.5 "GDB with a Tandem ST2000," page 137.

target vxworks *machinename*
: A VxWorks system, attached via TCP/IP. The argument *machinename* is the target system's machine name or IP address. See Section 13.4.6 "GDB and VxWorks," page 137.

`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. For example: using `target remote` *dev* with GDB standard remote protocol.

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

13.3 Choosing target byte order

You can now choose which byte order to use with a target system.

`set endian auto`
: Tells GDB to use the byte order associated with the executable.

`show endian`
: Print the current setting for byte order.

If you have no executable, or if the current setting does not match your configuration, you may need to use `set endian big` or `set endian little`.

`set endian big`
: Tells GDB that you are using a big endian chip. For example, GDB knows that a 68K is always a big endian chip, so you don't need to specify it. However, with a bi-endian chip such as a MIPS, you may need to tell GDB what to expect.

`set endian little`
: Tells GDB that you are using a little endian chip. For example, GDB knows that an X86 is always a little endian chip, so

you don't need to specify it. However, with a bi-endian chip such as a MIPS, you may need to tell GDB what to expect.

Warning: Currently, only embedded MIPS configurations support dynamic selection of target byte order.

13.4 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.

13.4.1 The GDB remote serial protocol

To debug a program running on another machine (the debugging *target* machine), you must first arrange for all the usual prerequisites for the program to run by itself. For example, for a C program, you need

1. A startup routine to set up the C runtime environment; these usually have a name like '`crt0`'. The startup routine may be supplied by your hardware supplier, or you may have to write your own.
2. You probably need a C subroutine library to support your program's subroutine calls, notably managing input and output.
3. A way of getting your program to the other machine—for example, a download program. These are often supplied by the hardware manufacturer, but you may have to write your own from hardware documentation.

The next step is to 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 looks like this:

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 Chapter 13 "Specifying a Debugging Target," page 119).

On the target,
: you must link with your program 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 Section 13.4.1.5 "Using the `gdbserver` program," page 129, 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.

These 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.

13.4.1.1 What the stub can do for you

The debugging stub for your architecture supplies these 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.

13.4.1.2 What you must do for the stub

The debugging stubs that come with GDB are set up for a particular chip architecture, but they have no information about the rest of your debugging target machine.

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

`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:

`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 interrupts themself without help from `exceptionHandler`.

`void flush_i_cache()`

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 this library routine is available:

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

This is the standard library function `memset` that 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.

13.4.1.3 Putting it all together

In summary, when your program is ready to debug, you must follow these steps.

1. Make sure you have the supporting low-level routines (see Section 13.4.1.2 “What you must do for the stub,” page 125):

   `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 used for this 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`’:

```
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`:

```
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 type the interrupt character (often `C-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 this prompt:

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

If you type `y`, 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 type `n`, GDB goes back to waiting.

13.4.1.4 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 checksum. 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.

These are the commands currently supported:

`g` Requests the values of CPU registers.

`G` Sets the values of CPU registers.

`m`*addr*`,`*count*
: Read *count* bytes at location *addr*.

`M`*addr*`,`*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.

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.

13.4.1.5 Using the `gdbserver` program

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

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

On the target machine,
: you need to have a copy of the program you want to debug. `gdbserver` does not need your program's symbol table, so you can strip the program if necessary to save space. GDB on the host system does all the symbol handling.

  To use the server, you must tell it how to communicate with GDB; the name of your program; and the arguments for your program. The syntax is:

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

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

```
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:

```
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`).[1] You must use the same port number with the host GDB `target remote` command.

On the GDB host machine,

you need an unstripped copy of your program, since GDB needs symbols and debugging information. Start up GDB as usual, using the name of the local copy of your program as the first argument. (You may also need the '`--baud`' option if the serial line is running at anything other than 9600 bps.) After that, use `target remote` to establish communications with `gdbserver`. Its argument is either a device name (usually a serial device, like '`/dev/ttyb`'), or a TCP port descriptor in the form *`host:PORT`*. For example:

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

communicates with the server via serial line '`/dev/ttyb`', and

```
(gdb) 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`'.

13.4.1.6 Using the `gdbserve.nlm` program

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

GDB and `gdbserve.nlm` communicate via a serial line, using the standard GDB remote serial protocol.

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:

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

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

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`'and communicate with GDB over serial port number 2 or board 1 using a 19200 bps connection:

```
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
```

communications with the server via serial line '`/dev/ttyb`'.

13.4.2 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 several 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 Section 13.3 “Commands for managing targets,” page 122.

13.4.2.1 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`') identifies the serial port 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 Section 13.3 “Commands for managing targets,” page 122).

13.4.2.2 Options for Nindy

These 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.

13.4.2.3 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.

13.4.3 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 a29k 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.

13.4.4 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, this development system is called the "EB29K". 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.

13.4.4.1 Communications setup

The next step is to set up the PC's port, by doing something like this 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, type the following at the DOS console:

```
C:\> 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 our 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 the following:

```
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/remote`'.

Using the `tip` or `cu` connection, change the DOS working directory to the directory containing a copy of your 29K program, then start the PC program EBMON (an EB29K control program supplied with your board by AMD). You should see an initial display from EBMON similar to the one that follows, ending with the EBMON prompt '`#`'—

```
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 (done in the example by typing `~.` at the EBMON prompt). EBMON keeps running, ready for GDB to take over.

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—of getting the 29K program from the Unix system to the PC; GDB does *not* download it over the serial line.

13.4.4.2 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:

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

Now you can use the `target` command:

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

In this example, we've assumed your program is in a file called '`myfoo`'. Note that 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`.

13.4.4.3 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.

13.4.5 GDB with a Tandem ST2000

To connect your ST2000 to the host system, see the manufacturer's manual. Once the 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 *dev* as a TCP connection (for example, to a serial line attached via a terminal concentrator) using the syntax *hostname*:*portnumber*.

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.

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

st2000 *command*
: Send a *command* to the STDBUG monitor. See the manufacturer's manual for available commands.

connect
: Connect the controlling terminal to the STDBUG command monitor. When you are done interacting with STDBUG, typing either of two character sequences gets you back to the GDB command prompt: RET~. (Return, followed by tilde and period) or RET~C-D (Return, followed by tilde and control-D).

13.4.6 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 program gdb is installed and executed on the Unix host. (It may be installed with the name vxgdb, to distinguish it from a GDB for debugging programs on the host itself.)

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)
```

13.4.6.1 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", type:

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

GDB displays messages like these:

```
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 Section 4.4 "Your program's environment," page 24); 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.

13.4.6.2 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 NFS mounted 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, type this on VxWorks:

```
-> 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 this:

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

13.4.6.3 Running tasks

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

```
(vxgdb) attach task
```

where *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.

13.4.7 GDB and Hitachi microprocessors

GDB needs to know these things to talk to your Hitachi SH, H8/300, or H8/500:

1. 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 300H. ('`target hms`' is the default when GDB is configured specifically for the Hitachi SH, H8/300, or H8/500.)
2. what serial device connects your host to your Hitachi board (the first serial device available on your host is the default).
3. what speed to use over the serial device.

13.4.7.1 Connecting to Hitachi boards

Use the special `gdb` command '`device` *port*' if you need to explicitly set the serial device. The default *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.

13.4.7.2 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 E7000:

`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.

13.4.7.3 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`.

13.4.8 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 these 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, this sequence connects to the target board through a serial port, and loads and runs a program called *prog* 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 these special commands for MIPS targets:

`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`. (These commands are *only* available 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.

13.4.9 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.

14 Controlling GDB

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, see Section 8.7 "Print settings," page 73; other settings are described here.

14.1 Prompt

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

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

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

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

14.2 Command editing

GDB reads its input commands via the *readline* interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or *vi*-style inline editing of commands, `csh`-like history substitution, and a storage and recall of command history across debugging sessions.

You may control the behavior of command line editing in GDB with the command `set`.

`set editing`
`set editing on`
: Enable command line editing (enabled by default).

`set editing off`
: Disable command line editing.

`show editing`
: Show whether command line editing is enabled.

14.3 Command history

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

`set history filename` *`fname`*
: Set the name of the GDB command history file to *`fname`*. This is the file where GDB reads an initial command history list, and where it writes the command history from this session when it exits. You can access this list through history expansion or through the history command editing characters listed below. This file defaults to the value of the environment variable `GDBHISTFILE`, or to '`./.gdb_history`' if this variable is not set.

`set history save`
`set history save on`
: Record command history in a file, whose name may be specified with the `set history filename` command. By default, this option is disabled.

`set history save off`
: Stop recording command history in a file.

`set history size` *`size`*
: Set the number of commands which GDB keeps in its history list. This defaults to the value of the environment variable `HISTSIZE`, or to 256 if this variable is not set.

History expansion assigns special meaning to the character `!`.

Since `!` is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the `set history expansion on` command, you may sometimes need to follow `!` (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities do not attempt substitution on the strings `!=` and `!(`, even when history expansion is enabled.

The commands to control history expansion are:

`set history expansion on`
`set history expansion`
: Enable history expansion. History expansion is off by default.

`set history expansion off`
: Disable history expansion.

The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with GNU Emacs or `vi` may wish to read it.

`show history`
`show history filename`
`show history save`
`show history size`
`show history expansion`
: These commands display the state of the GDB history parameters. `show history` by itself displays all four states.

`show commands`
: Display the last ten commands in the command history.

`show commands` *n*
: Print ten commands centered on command number *n*.

`show commands +`
: Print ten commands just after the commands last printed.

14.4 Screen size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type `RET` when you want to continue the output, or `q` to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally GDB knows the size of the screen from the 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.

14.5 Numbers

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with '0', decimal numbers end with '.', and hexadecimal numbers begin with '0x'. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers—when no particular format is specified—is base 10. You can change the default base for both input and output with the `set radix` command.

`set input-radix` *base*
: Set the default base for numeric input. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix; for example, any of

```
set radix 012
set radix 10.
set radix 0xa
```

sets the base to decimal. On the other hand, 'set radix 10' leaves the radix unchanged no matter what it was.

`set output-radix` *base*
: Set the default base for numeric display. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix.

`show input-radix`
: Display the current default base for numeric input.

`show output-radix`
: Display the current default base for numeric display.

14.6 Optional warnings and messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the `set verbose` command. This makes GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by `set verbose` are those which announce that the symbol table for a source file is being read; see `symbol-file` in Section 12.1 "Commands to specify files," page 111.

`set verbose on`
: Enables GDB output of certain informational messages.

`set verbose off`
: Disables GDB output of certain informational messages.

`show verbose`
: Displays whether `set verbose` is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful (see Section 12.2 “Errors reading symbol files,” page 115).

`set complaints` *`limit`*
: Permits GDB to output *`limit`* complaints about each type of unusual symbols before becoming silent about the problem. Set *`limit`* to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

`show complaints`
: Displays how many symbol complaints GDB is permitted to produce.

By default, GDB is cautious, and asks what sometimes seems to be a lot of stupid questions to confirm certain commands. For example, if you try to run a program which is already running:

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

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

`set confirm off`
: Disables confirmation requests.

`set confirm on`
: Enables confirmation requests (the default).

`show confirm`
: Displays state of confirmation requests.

15 Canned Sequences of Commands

Aside from breakpoint commands (see Section 5.1.7 "Breakpoint command lists," page 43), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

15.1 User-defined commands

A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command. This is done with the `define` command. User commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command via *$arg0...$arg9*. A trivial example:

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

To execute the command use:

```
adder 1 2 3
```

This defines the command `adder`, which prints the sum of its three arguments. Note the arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior functions calls.

`define` *commandname*
: Define a command named *commandname*. If there is already a command by that name, you are asked to confirm that you want to redefine it.

: The definition of the command is made up of other GDB command lines, which are given following the `define` command. The end of these commands is marked by a line containing `end`.

`if`
: Takes a single argument, which is an expression to evaluate. It is followed by a series of commands that are executed only if the expression is true (nonzero). There can then optionally be a line `else`, followed by a series of commands that are only executed if the expression was false. The end of the list is marked by a line containing `end`.

`while`
: The syntax is similar to `if`: the command takes a single argument, which is an expression to evaluate, and must be followed by the commands to execute, one per line, terminated by an `end`. The commands are executed repeatedly as long as the expression evaluates to true.

`document commandname`
: Document the user-defined command *commandname*, so that it can be accessed by `help`. The command *commandname* must already be defined. This command reads lines of documentation just as `define` reads the lines of the command definition, ending with `end`. After the `document` command is finished, `help` on command *commandname* displays the documentation you have written.

  You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

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

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

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

15.2 User-defined command hooks

You may define *hooks*, which are a special kind of user-defined command. Whenever you run the command '`foo`', if the user-defined command '`hook-foo`' exists, it is executed (with no arguments) before that command.

In addition, a pseudo-command, '`stop`' exists. Defining ('`hook-stop`') makes the associated commands execute every time execution stops in your program: before breakpoint commands are run, displays are printed, or the stack frame is printed.

For example, to ignore `SIGALRM` signals while single-stepping, but treat them normally during normal execution, you could define:

```
define hook-stop
handle SIGALRM nopass
end
```

```
define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

You can define a hook for any single-word command in GDB, but not for command aliases; you should define a hook for the basic command name, e.g. `backtrace` rather than `bt`. If an error occurs during the execution of your hook, execution of GDB commands stops and GDB issues a prompt (before the command that you actually typed had a chance to run).

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

15.3 Command files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting with `#`) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When you start GDB, it automatically executes commands from its *init files*. These are files named '`.gdbinit`'. 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 Section 2.1.2 "Choosing modes," page 11.

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.

15.4 Commands for controlled output

During the execution of a command file or a user-defined command, normal GDB output is suppressed; the only output that appears is what is explicitly printed by the commands in the definition. This section describes three commands useful for generating exactly the output you want.

`echo` *text*

Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as '`\n`' to print a newline. **No newline is printed unless you specify one.** In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments. To print '` and foo = `', use the command '`echo \ and foo = \ `'.

A backslash at the end of *text* can be used, as in C, to continue the command onto subsequent lines. For example,

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

produces the same output as

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

`output` *expression*

Print the value of *expression* and nothing but that value: no newlines, no '`$`*nn* `= `'. The value is not entered in the value history either. See Section 8.1 "Expressions," page 65, for more information on expressions.

`output/`*fmt expression*

Print the value of *expression* in format *fmt*. You can use the same formats as for `print`. See Section 8.4 "Output formats," page 68, for more information.

`printf` *string*, *expressions*. . .

Print the values of the *expressions* under the control of *string*. The *expressions* are separated by commas and may be either numbers or pointers. Their values are printed as specified by *string*, exactly as if your program were to execute the C subroutine

```
printf (string, expressions...);
```

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences that you can use in the format string are the simple ones that consist of backslash followed by a letter.

16 Using GDB under GNU Emacs

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

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

Using GDB under Emacs is just like using GDB normally except for two things:

- All "terminal" input and output goes through the Emacs buffer.

This applies both to GDB commands and their output, and to the input and output done by the program you are debugging.

This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

All the facilities of Emacs' Shell mode are available for interacting with your program. In particular, you can send signals the usual way—for example, `C-c C-c` for an interrupt, `C-c C-z` for a stop.

- GDB displays source code through Emacs.

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

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

> *Warning:* If the directory where your program resides is not your current directory, it can be easy to confuse Emacs about the location of the source files, in which case the auxiliary display buffer does not appear to show your source. GDB can find programs by searching your environment's `PATH` variable, so the GDB input and output session proceeds normally; but Emacs does not get enough information back from GDB to locate the source files in this situation. To avoid this problem, either start GDB mode from the directory where your program resides, or specify an absolute file name when prompted for the `M-x gdb` argument.
>
> A similar confusion can result if you use the GDB `file` command to switch to debugging a program in some other location, from an existing GDB buffer in Emacs.

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

```
(setq gdb-command-name "mygdb")
```

(preceded by `ESC ESC`, or typed in the `*scratch*` buffer, or in your '`.emacs`' file) makes Emacs call the program named "`mygdb`" instead.

In the GDB I/O buffer, you can use these special Emacs commands in addition to the standard Shell mode commands:

`C-h m`
: Describe the features of Emacs' GDB Mode.

`M-s`
: Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

`M-n`
: Execute to next source line in this function, skipping all function calls, like the GDB `next` command. Then update the display window to show the current file and location.

`M-i`
: Execute one instruction, like the GDB `stepi` command; update display window accordingly.

`M-x gdb-nexti`
: Execute to next instruction, using the GDB `nexti` command; update display window accordingly.

`C-c C-f`
: Execute until exit from the selected stack frame, like the GDB `finish` command.

`M-c`
: Continue execution of your program, like the GDB `continue` command.

  Warning: In Emacs v19, this command is `C-c C-p`.

`M-u`
: Go up the number of frames indicated by the numeric argument (see section "Numeric Arguments" in *The* GNU *Emacs Manual*), like the GDB `up` command.

  Warning: In Emacs v19, this command is `C-c C-u`.

`M-d`
: Go down the number of frames indicated by the numeric argument, like the GDB `down` command.

  Warning: In Emacs v19, this command is `C-c C-d`.

`C-x &`
: Read the number where the cursor is positioned, and insert it at the end of the GDB I/O buffer. For example, if you wish to disassemble code around an address that was displayed earlier, type `disassemble`; then move the cursor to the address display, and pick up the argument for `disassemble` by typing `C-x &`.

You can customize this further by defining elements of the list `gdb-print-command`; once it is defined, you can format or otherwise process numbers picked up by `C-x &` before they are inserted. A numeric argument to `C-x &` indicates that you wish special formatting, and also acts as an index to pick an element of the list. If the list element is a string, the number to be inserted is formatted using the Emacs function `format`; otherwise the number is passed as an argument to the corresponding list element.

In any source file, the Emacs command `C-x SPC` (`gdb-break`) tells GDB to set a breakpoint on the source line point is on.

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command `f` in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers. If you add or delete lines from the text, the line numbers that GDB knows cease to correspond properly with the code.

17 Reporting Bugs in GDB

Your bug reports play an essential role in making GDB reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of GDB work better. Bug reports are your contribution to the maintenance of GDB.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

17.1 Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the debugger gets a fatal signal, for any input whatever, that is a GDB bug. Reliable debuggers never crash.
- If GDB produces an error message for valid input, that is a bug.
- 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.

17.2 How to report bugs

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 that 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
545 Tech Square
Cambridge, MA 02139
```

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 these things:

- The version of GDB. GDB announces it if you start with no arguments; you can also print it at any time using `show version`.

  Without this, we will not know whether there is any point in looking for the bug in the current version of GDB.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile GDB—e.g. "gcc–2.0".
- What compiler (and its version) was used to compile the program you are debugging—e.g. "gcc–2.0".
- 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.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

  However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

  A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GDB it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

  And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

Appendix A Command Line Editing

This text describes GNU's command line editing interface.

A.1 Introduction to Line Editing

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

The text `C-K` is read as 'Control-K' and describes the character produced when the Control key is depressed and the `K` key is struck.

The text `M-K` is read as 'Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the `K` key is struck. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `K`. Either process is known as *metafying* the `K` key.

The text `M-C-K` is read as 'Meta-Control-k' and describes the character produced by *metafying* `C-K`.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see Section A.3 "Readline Init File," page 168, for more info).

A.2 Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RETURN`. You do not have to be at the end of the line to press `RETURN`; the entire line is accepted regardless of the location of the cursor within the line.

A.2.1 Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use `DEL` to back up, and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type `C-B` to move the cursor to the left, and then correct your mistake. Aftwerwards, 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.

`DEL` 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.

A.2.2 Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-B`, `C-F`, `C-D`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-A` Move to the start of the line.

`C-E` Move to the end of the line.

`M-F` Move forward a word.

`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.

A.2.3 Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* 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.

Here is the list of commands for killing text.

`C-K` Kill the text from the current cursor position to the end of the line.

`M-D` Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`M-DEL` Kill from the cursor 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-DEL` because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking is

`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.

A.2.4 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type `M-- C-K`.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' 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 type `M-1 0 C-D`.

A.3 Readline Init File

Although the Readline library comes with a set of 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.

A.3.1 Readline Init Syntax

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

Variable Settings

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

```
set editing-mode vi
```

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

`editing-mode`

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

`horizontal-scroll-mode`

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

`mark-modified-lines`

This variable when set to `On`, says to display an asterisk ('`*`') at the starts of history lines which have been modified. This variable is off by default.

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

Key Bindings

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

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the '`~/.inputrc`' file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: *function-name* or *macro*
: *keyname* is the name of a key spelled out in English. For example:

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

In the above example, `C-U` is bound to the function `universal-argument`, and `C-O` is bound to run the macro expressed on the right hand side (that is, to insert the text '`>&output`' into the line).

"*keyseq*": *function-name* or *macro*
: *keyseq* differs from *keyname* above in that strings denoting an entire key sequence can be specified. Simply place the key sequence in double quotes. GNU Emacs style key escapes can be used, as in the following example:

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

In the above example, `C-U` is bound to the function `universal-argument` (just as it was in the first example), `C-X C-R` is bound to the function `re-read-init-file`, and `ESC [ 1 1 ~` is bound to insert the text '`Function Key 1`'.

A.3.1.1 Commands For Moving

`beginning-of-line (C-A)`
: Move to the start of the current line.

`end-of-line (C-E)`
: Move to the end of the line.

`forward-char (C-F)`
: Move forward a character.

`backward-char (C-B)`
: Move back a character.

`forward-word (M-F)`
: Move forward to the end of the next word.

`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.

A.3.1.2 Commands For Manipulating The History

`accept-line (Newline, Return)`
: Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-P)`
: Move 'up' through the history list.

`next-history (C-N)`
: Move 'down' through the history list.

`beginning-of-history (M-<)`
: Move to the first line in the history.

`end-of-history (M->)`
: Move to the end of the input history, i.e., the line 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.

A.3.1.3 Commands For Changing Text

`delete-char (C-D)`
: Delete the character under the cursor. If the cursor is at the beginning of the line, 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 arg 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 args 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.

A.3.1.4 Killing And Yanking

`kill-line (C-K)`
: Kill the text from the current cursor position to the end of the line.

`backward-kill-line ()`
: 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-DEL)`
: Kill the word behind the cursor.

`unix-line-discard (C-U)`
: Do what `C-U` used to do in Unix line input. We save the killed text on the kill-ring, though.

`unix-word-rubout (C-W)`
: Do what `C-W` used to do 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.

A.3.1.5 Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`
: Add this digit to the argument already accumulating, or start a new argument. `M--` starts a negative argument.

`universal-argument ()`
: Do what `C-U` does in emacs. By default, this is not bound.

A.3.1.6 Letting Readline Type For You

`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.

A.3.1.7 Some Miscellaneous Commands

`re-read-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.

A.3.2 Readline Vi Mode

While the Readline library does not have a full set of Vi editing functions, it does contain enough to allow simple editing of the line.

In order to switch interactively between 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'. Pressing `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.

Appendix B Using History Interactively

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

B.1 History Interaction

The History library provides a history expansion feature that is similar to the history expansion in Csh. The following text describes the sytax that you use to manipulate the history information.

History expansion takes place in two parts. The first is to determine which line from the previous history should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is called the *event*, and the portions of that line that are acted upon are called *words*. The line is broken into words in the same fashion that the Bash shell does, so that several English (or Unix) words surrounded by quotes are considered as one word.

B.1.1 Event Designators

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

`!` Start a history subsititution, 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*.

B.1.2 Word Designators

A `:` separates the event specification 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.

B.1.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`.

`#` The entire command line typed so far. This means the current command, not the previous command, so it really isn't a word designator, and doesn't belong in this section.

`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.

Appendix C 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[1]. 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 on-line information and a printed manual. You can use one of the Info formatting commands to create the on-line version of the documentation and TEX (or texi2roff) to typeset the printed version.

GDB includes an already formatted copy of the on-line 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 ('gdb-', in the case of version), 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,

[1] In 'gdb-/gdb/refcard.ps' of the version release.

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 the '`gdb`' subdirectory of the main source directory (for example, to '`gdb-/gdb`') and then type:

```
make gdb.dvi
```

Appendix D 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.[1]

The GDB distribution includes all the source code you need for GDB in a single directory, whose name is usually composed by appending the version number to '`gdb`'.

For example, the GDB version distribution is in the '`gdb-`' directory. That directory contains:

`gdb-/configure` (and supporting files)
: script for configuring GDB and all its supporting libraries

`gdb-/gdb`
: the source specific to GDB itself

`gdb-/bfd`
: source for the Binary File Descriptor library

`gdb-/include`
: GNU include files

`gdb-/libiberty`
: source for the '`-liberty`' free software library

`gdb-/opcodes`
: source for the library of opcode tables and disassemblers

`gdb-/readline`
: source for the GNU command-line interface

`gdb-/glob`
: source for the GNU filename pattern-matching subroutine

`gdb-/mmalloc`
: source for the GNU memory-mapped malloc package

The simplest way to configure and build GDB is to run `configure` from the '`gdb-`*`version-number`*' source directory, which in this example is the '`gdb-`' directory.

First switch to the '`gdb-`*`version-number`*' source directory if you are not already in it; then run `configure`. Pass the identifier for the platform on which GDB will run as an argument.

For example:

---

[1] If you have a more recent version of GDB than , look at the '`README`' file in the sources; we may have improved the installation procedures since publishing this manual.

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

where *host* is an identifier such as 'sun4' or 'decstation', that identifies the platform where GDB will run. (You can often leave off *host*; configure tries to guess the correct value by examining your system.)

Running 'configure *host*' and then running make builds the 'bfd', 'readline', 'mmalloc', and 'libiberty' libraries, then gdb itself. The configured source files, and the binaries, are left in the corresponding source directories.

configure is a Bourne-shell (/bin/sh) script; if your system does not recognize this automatically when you run a different shell, you may need to run sh on it explicitly:

```
sh configure host
```

If you run configure from a directory that contains source directories for multiple libraries or programs, such as the 'gdb-' source directory for version , 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 , type the following to configure only the bfd subdirectory:

```
cd gdb-/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.

D.1 Compiling GDB in another directory

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

To build gdb in a separate directory, run configure with the '--srcdir' option to specify where to find the source. (You also need

to specify a path to find `configure` itself from your working directory. If the path to `configure` would be the same as the argument to '`--srcdir`', you can leave out the '`--srcdir`' option; it is assumed.)

For example, with version , you can build GDB in a separate directory for a Sun 4 like this:

```
cd gdb-
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-/configure sun4
make
```

When `configure` builds a configuration using a remote source directory, it creates a tree for the binaries with the same structure (and using the same names) as the tree under the source directory. In the example, you'd find the Sun 4 library '`libiberty.a`' in the directory '`gdb-sun4/libiberty`', and GDB itself in '`gdb-sun4/gdb`'.

One popular reason to build several GDB configurations in separate directories is to configure GDB for cross-compiling (where GDB runs on one machine—the *host*—while debugging programs that run on another machine—the *target*). You specify a cross-debugging target by giving the '`--target=`*`target`*' option to `configure`.

When you run `make` to build a program or library, you must run it in a configured directory—whatever directory you were in when you called `configure` (or one of its subdirectories).

The `Makefile` that `configure` generates in each source directory also runs recursively. If you type `make` in a source directory such as '`gdb-`' (or in a separate configured directory configured with '`--srcdir=`*`dirname`*`/gdb-`'), 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.

D.2 Specifying names for hosts and targets

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

`architecture-vendor-os`

For example, you can use the alias `sun4` as a *host* argument, or as the value for *target* in a `--target=target` option. The equivalent full name is '`sparc-sun-sunos4`'.

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

```
% sh config.sub 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 ('`gdb-`', for version).

D.3 `configure` options

Here is a summary of the `configure` options and arguments that are most often useful for building GDB. `configure` also has several other options not listed here. See Info file '`configure.info`', node '`What Configure Does`', for a full explanation of `configure`.

```
configure [--help]
          [--prefix=dir]
          [--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`

Warning: using this option requires GNU `make`, or another `make` that implements the `VPATH` feature.
Use this option to make configurations in directories separate from the GDB source directories. Among other things, you can use this to build (or maintain) several configurations simultaneously, in separate directories. `configure` writes configuration specific files in the current directory, but arranges for them to use the source in the directory *dirname*. `configure` creates directories under the working directory in parallel to the source directories below *dirname*.

`--norecursion`

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

`--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.

Index

#

.... 15
in Modula-2 102

$

$ 78
$$ 78
$_ 80
$_ and info breakpoints 36
$_ and info line 63
$_, $__, and value history 71
$__ 80
$bpnum 34
$cdir 62
$cwd 62

.

. ... 101
.esgdbinit 153
'.gdbinit' 153
.os68gdbinit 153
.vxgdbinit 153

/

/proc 28

:

:: 67, 101

@

@ 67

{

{type} 66

A

a.out and C++ 93
abbreviation 15
active targets 119
add-shared-symbol-file 114
add-symbol-file 114
AMD 29K register stack 82
AMD EB29K 121
AMD29K via UDI 133
arguments (to your program) 24
artificial array 67
assembly instructions 63
assignment 107
attach 27
automatic display 71
automatic thread selection 31
awatch 38

B

b 34
backtrace 54
break 34
break ... thread threadno 50
break in overloaded functions 95
breakpoint commands 43
breakpoint conditions 42
breakpoint numbers 33
breakpoint on memory address 33
breakpoint on variable modification ... 33
breakpoint subroutine, remote 125
breakpoints 33
breakpoints and threads 50
bt 54
bug criteria 161
bug reports 161
bugs in GDB 161

C

c 46
C and C++ 90
C and C++ checks 95
C and C++ constants 93
C and C++ defaults 94
C and C++ operators 91
C++ 90
C++ and object formats 93
C++ exception handling 96
C++ scope resolution 67
C++ support, not in coff 93

C++ symbol decoding style 77
C++ symbol display 96
`call` 110
call overloaded functions 94
call stack 53
calling functions 110
calling make 13
casts, to view memory 66
`catch` 39
catch exceptions 57
`cd` 26
`cdir` 62
checks, range 89
checks, type 88
checksum, for GDB remote 128
choosing target byte order 122
`clear` 40
clearing breakpoints, watchpoints 40
coff versus C++ 93
colon, doubled as scope operator 101
colon-colon 67
command files 152, 153
command line editing 145
`commands` 43
commands for C++ 95
commands to STDBUG (ST2000) 137
comment 15
compilation directory 62
`complete` 18
completion 16
completion of quoted strings 16
`condition` 42
conditional breakpoints 42
configuring GDB 179
confirmation 149
connect (to STDBUG) 137
`continue` 46
continuing 45
continuing threads 51
control C, and remote debugging 125
controlling terminal 26
convenience variables 79
`core` 113
core dump file 111
`core-file` 113
CPU simulator 142
crash of debugger 161
current directory 62
current thread 29
`cwd` 62

D

`d` 40
debugger crash 161
debugging optimized code 21
debugging stub, example 128
debugging target 119
`define` 151
`delete` 40
delete breakpoints 40
`delete display` 72
deleting breakpoints, watchpoints 40
demangling 77
`detach` 27
`device` 139
`dir` 61
directories for source files 61
`directory` 61
directory, compilation 62
directory, current 62
`dis` 41
`disable` 41
`disable breakpoints` 40, 41
`disable display` 72
`disassemble` 63
`display` 72
display of expressions 71
`do` 56
`document` 151
documentation 177
`down` 56
`down-silently` 56
download to H8/300 or H8/500 114
download to Hitachi SH 114
download to Nindy-960 114
download to VxWorks 138
dynamic linking 114

E

`eb.log` 136
EB29K board 134
`EBMON` 135
`echo` 154
ecoff and C++ 93
editing 145
`editing-mode` 168

elf/dwarf and C++ 93
elf/stabs and C++ 93
else 151
Emacs 157
enable 41
enable breakpoints 40, 41
enable display 72
end 43
entering numbers 148
environment (of your program) 24
error on valid input 161
event designators 175
examining data 65
examining memory 69
exception handlers 39, 57
exceptionHandler 126
exec-file 111
executable file 111
exiting GDB 12
expansion 175
expressions 65
expressions in C or C++ 90
expressions in C++ 93
expressions in Modula-2 96

F

f 55
fatal signal 161
fatal signals 49
fg 46
file 111
finish 47
flinching 149
floating point 83
floating point registers 81
floating point, MIPS remote 141
flush_i_cache 126
focus of debugging 29
foo 116
fork, debugging programs which call .. 31
format options 73
formatted output 68
Fortran 1
forward-search 61
frame 53
frame 54, 55
frame number 53
frame pointer 53
frameless execution 54

G

g++ 90
GDB bugs, reporting 161
GDB reference card 177
GDBHISTFILE 146
gdbserve.nlm 131
gdbserver 129
getDebugChar 125
gnu C++ 90
gnu Emacs 157

H

h 17
H8/300 or H8/500 download 114
H8/300 or H8/500 simulator 142
handle 49
handle_exception 124
handling signals 49
hbreak 35
help 17
help target 120
help user-defined 152
heuristic-fence-post (MIPS) 58
history expansion 146
history file 146
history number 78
history save 146
history size 146
history substitution 146
Hitachi SH download 114
Hitachi SH simulator 142
horizontal-scroll-mode 168

I

i 19
i/o 26
i386 124
i386-stub.c 124
i960 132
if 151
ignore 43
ignore count (of breakpoint) 43
INCLUDE_RDB 137
info 19
info address 103

info all-registers 81
info args 57
info breakpoints 36
info catch 57
info display 72
info f 56
info files 115
info float 83
info frame 56, 87
info functions 104
info line 62
info locals 57
info proc 28
info proc id 28
info proc mappings 28
info proc status 29
info proc times 28
info program 33
info registers 81
info s 55
info set 19
info share 115
info sharedlibrary 115
info signals 49
info source 87, 104
info sources 104
info stack 55
info target 115
info terminal 26
info threads 30
info types 104
info variables 105
info watchpoints 38
inheritance 96
init file 153
init file name 153
initial frame 53
innermost frame 53
inspect 65
installation 179
instructions, assembly 63
Intel 124
interaction, readline 165
internal GDB breakpoints 37
interrupt 12
interrupting remote programs 128
interrupting remote targets 125
invalid input 161

J

jump 108

K

kill 28

L

l 59
languages 85
latest breakpoint 34
leaving GDB 12
linespec 60
list 59
listing machine instructions 63
load *filename* 113
log file for EB29K 136

M

m680x0 124
m68k-stub.c 124
machine instructions 63
maint info breakpoints 37
maint print psymbols 105
maint print symbols 105
make 13
mapped 112
mark-modified-lines 168
member functions 94
memory models, H8/500 140
memory tracing 33
memory, viewing as typed object ... 66
memory-mapped symbol file 112
memset 126
MIPS boards 141
MIPS remote floating point 141
MIPS remotedebug protocol 142
MIPS stack 58
Modula-2 96
Modula-2 built-ins 98
Modula-2 checks 101
Modula-2 constants 99
Modula-2 defaults 100
Modula-2 operators 97
Modula-2, deviations from 100
Motorola 680x0 124
multiple processes 31
multiple targets 119

multiple threads 29

N

n 47
names of symbols 103
namespace in C++ 94
negative breakpoint numbers 37
New *systag* 30
next 47
nexti 48
ni 48
Nindy 132
number representation 148
numbers for breakpoints 33

O

object formats and C++ 93
online documentation 17
optimized code, debugging 21
outermost frame 53
output 154
output formats 68
overloading 45
overloading in C++ 95

P

packets, reporting on stdout 129
partial symbol dump 105
patching binaries 110
path 24
pauses in output 147
pipes 24
pointer, finding referent 74
prefer-visible-bell 169
print 65
print settings 73
printf 154
printing data 65
process image 28
processes, multiple 31
prompt 145
protocol, GDB remote serial 128
ptype 103
putDebugChar 125
pwd 26

Q

q 12
quit 12
quotes in commands 16
quoting names 103

R

raise exceptions 39
range checking 89
rbreak 36
reading symbols immediately 112
readline 145
readnow 112
redirection 26
reference card 177
reference declarations 94
register stack, AMD29K 82
registers 81
regular expression 36
reloading symbols 105
remote connection without stubs 129
remote debugging 123
remote programs, interrupting 128
remote serial debugging summary 127
remote serial debugging, overview 123
remote serial protocol 128
remote serial stub 124
remote serial stub list 124
remote serial stub, initialization 124
remote serial stub, main routine 124
remote stub, example 128
remote stub, support routines 125
remotedebug, MIPS protocol 142
repeating commands 15
reporting bugs in GDB 161
reset 133
response time, MIPS debugging 58
resuming execution 45
RET 15
retransmit-timeout, MIPS protocol 142
return 109
returning from a function 109
reverse-search 61
run 23
running 23
running 29K programs 134
running VxWorks tasks 139

rwatch 38

S

s 46
saving symbol table 112
scope 101
search 61
searching 61
section 114
select-frame 54
selected frame 53
serial connections, debugging 129
serial device, Hitachi micros 139
serial line speed, Hitachi micros 139
serial line, target remote 127
serial protocol, GDB remote 128
set 19
set args 24
set check 88, 89
set check range 89
set check type 88
set complaints 149
set confirm 149
set demangle-style 77
set editing 145
set endian auto 122
set endian big 122
set endian little 122
set environment 25
set gnutarget 120
set height 147
set history expansion 146
set history filename 146
set history save 146
set history size 146
set input-radix 148
set language 86
set listsize 59
set machine 140
set memory *mod* 140
set mipsfpu 141
set output-radix 148
set print address 73
set print array 75
set print asm-demangle 77
set print demangle 77
set print elements 75
set print max-symbolic-offset .. 74
set print null-stop 75
set print object 78
set print pretty 75
set print sevenbit-strings 76
set print symbol-filename 74
set print union 76
set print vtbl 78
set prompt 145
set remotedebug 129, 142
set retransmit-timeout 142
set rstack_high_address 82
set symbol-reloading 105
set timeout 142
set variable 107
set verbose 148
set width 147
set write 110
set_debug_traps 124
setting variables 107
setting watchpoints 38
share 115
shared libraries 115
sharedlibrary 115
shell 13
shell escape 13
show 19
show args 24
show check range 89
show check type 88
show commands 147
show complaints 149
show confirm 149
show convenience 80
show copying 19
show demangle-style 78
show directories 62
show editing 145
show endian 122
show environment 25
show gnutarget 120
show height 147
show history 147
show input-radix 148
show language 87
show listsize 59
show machine 140
show mipsfpu 141
show output-radix 148

show paths 25
show print address 74
show print array 75
show print asm-demangle 77
show print demangle 77
show print elements 75
show print max-symbolic-offset 74
show print object 78
show print pretty 76
show print sevenbit-strings 76
show print symbol-filename 74
show print union 76
show print vtbl 78
show prompt 145
show remotedebug 129, 142
show retransmit-timeout 142
show rstack_high_address 82
show symbol-reloading 105
show timeout 142
show user 152
show values 79
show verbose 149
show version 19
show warranty 19
show width 147
show write 110
si 48
signal 109
signals 49
silent 44
sim 143
simulator 142
simulator, H8/300 or H8/500 142
simulator, Hitachi SH 142
simulator, Z8000 142
size of screen 147
source 153
source path 61
sparc-stub.c 124
speed 139
ST2000 auxiliary commands 137
st2000 *cmd* 137
stack frame 53
stack on MIPS 58
stacking targets 119
starting 23
STDBUG commands (ST2000) 137
step 46
stepi 48
stepping 45
stopped threads 51
stub example, remote debugging 128
stupid questions 149
switching threads 29
switching threads automatically 31
symbol decoding style, C++ 77
symbol dump 105
symbol names 103
symbol overloading 45
symbol table 111
symbol-file 111
symbols, reading immediately 112

T

target 119
target amd-eb 121
target array 122
target byte order 122
target core 120
target cpu32bug 121
target e7000 140
target est 122
target exec 120
target hms 121
target mips *port* 141
target nindy 121
target op50n 122
target remote 120
target rom68k 122
target sim 121, 143
target sparclite 122
target st2000 121
target udi 121
target vxworks 121
target w89k 122
tbreak 35
TCP port, target remote 128
terminal 26
thbreak 35
this 94
thread apply 30
thread breakpoints 50
thread identifier (GDB) 30
thread identifier (system) 30
thread number 30

`thread` *`threadno`* 30
threads and watchpoints 38
threads of execution 29
threads, automatic switching 31
threads, continuing 51
threads, stopped 51
`timeout`, MIPS protocol 142
`toggle-editing-mode` 173
`tty` 26
type casting memory 66
type checking 88
type conversions in C++ 94

U

`u` 47
`udi` 134
UDI 133
`undisplay` 72
unknown address, locating 69
`unset environment` 25
`until` 47
`up` 56
`up-silently` 56
user-defined command 151

V

value history 78
variable name conflict 67
variable values, wrong 67
variables, setting 107
version number 19
`vi` style command editing 173
VxWorks 137

W

`watch` 38
watchpoints 33
watchpoints and threads 38
`whatis` 103
`where` 55
`while` 151
wild pointer, interpreting 74
word completion 16
working directory 62
working directory (of your program) .. 26
working language 85
writing into corefiles 110
writing into executables 110
wrong values 67

X

`x` 69
xcoff and C++ 93

Z

Z8000 simulator 142

The body of this manual is set in
pncr at 10.95pt,
with headings in **pncb at 10.95pt**
and examples in `pcrr`.
pncri at 10.95pt,
pncb at 10.95pt, and
`pcrro`
are used for emphasis.

The GNU C++ Iostream Library

Reference Manual for `libio` Version 0.64

Per Bothner bothner@cygnus.com
Roland Pesch pesch@cygnus.com

Copyright © 1993 Free Software Foundation, Inc.

`libio` includes software developed by the University of California, Berkeley.

`libio` uses floating-point software written by David M. Gay, which includes the following notice:

> The author of this software is David M. Gay.
>
> Copyright (c) 1991 by AT&T.
>
> Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.
>
> THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

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

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

Table of Contents

1 Introduction 1

1.1 Licensing terms for `libio` 1
1.2 Acknowledgements 1

2 Operators and Default Streams 3

3 Stream Classes 5

3.1 Shared properties: class `ios` 5
3.1.1 Checking the state of a stream 5
3.1.2 Choices in formatting 7
3.1.3 Changing stream properties using manipulators 10
3.1.4 Extended data fields 11
3.1.5 Synchronizing related streams 12
3.1.6 Reaching the underlying `streambuf` 12
3.2 Managing output streams: class `ostream` 13
3.2.1 Writing on an `ostream` 13
3.2.2 Repositioning an `ostream` 14
3.2.3 Miscellaneous `ostream` utilities 14
3.3 Managing input streams: class `istream` 15
3.3.1 Reading one character 15
3.3.2 Reading strings 16
3.3.3 Repositioning an `istream` 17
3.3.4 Miscellaneous `istream` utilities 18
3.4 Input and output together: class `iostream` 19

4 Classes for Files and Strings 21

4.1 Reading and writing files 21
4.2 Reading and writing in memory 23

5 Using the `streambuf` Layer 25

5.1 Areas of a `streambuf` 25
5.2 Simple output re-direction by redefining `overflow` 26
5.3 C-style formatting for `streambuf` objects 28
5.4 Wrappers for C `stdio` 28
5.5 Reading/writing from/to a pipe 29
5.6 Backing up 29
5.7 Forwarding I/O activity 31

6 C Input and Output **33**

Index **35**

1 Introduction

The iostream classes implement most of the features of AT&T version 2.0 iostream library classes, and most of the features of the ANSI X3J16 library draft (which is based on the AT&T design).

This manual is meant as a reference; for tutorial material on iostreams, see the corresponding section of any recent popular introduction to C++.

1.1 Licensing terms for `libio`

Since the `iostream` classes are so fundamental to standard C++, the Free Software Foundation has agreed to a special exception to its standard license, when you link programs with `libio.a`:

> As a special exception, if you link this library with files compiled with a GNU compiler to produce an executable, this does not cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

The code is under the GNU General Public License (version 2) for all other purposes than linking with this library; that means that you can modify and redistribute the code as usual, but remember that if you do, your modifications, and anything you link with the modified code, must be available to others on the same terms.

These functions are also available as part of the `libg++` library; if you link with that library instead of `libio`, the GNU Library General Public License applies.

1.2 Acknowledgements

Per Bothner wrote most of the `iostream` library, but some portions have their origins elsewhere in the free software community. Heinz Seidl wrote the IO manipulators. The floating-point conversion software is by David M. Gay of AT&T. Some code was derived from parts of BSD 4.4, which was written at the University of California, Berkeley.

The iostream classes are found in the `libio` library. An early version was originally distributed in `libg++`, and they are still included there as well, for convenience if you need other `libg++` classes. Doug Lea was the original author of `libg++`, and some of the file-management code still in `libio` is his.

Various people found bugs or offered suggestions. Hongjiu Lu worked hard to use the library as the default stdio implementation for Linux, and has provided much stress-testing of the library.

2 Operators and Default Streams

The GNU iostream library, ‘`libio`’, implements the standard input and output facilities for C++. These facilities are roughly analogous (in their purpose and ubiquity, at least) with those defined by the C ‘`stdio`’ functions.

Although these definitions come from a library, rather than being part of the “core language”, they are sufficiently central to be specified in the latest working papers for C++.

You can use two operators defined in this library for basic input and output operations. They are familiar from any C++ introductory textbook: `<<` for output, and `>>` for input. (Think of data flowing in the direction of the “arrows”.)

These operators are often used in conjunction with three streams that are open by default:

`ostream` **cout** — Variable

The standard output stream, analogous to the C `stdout`.

`istream` **cin** — Variable

The standard input stream, analogous to the C `stdin`.

`ostream` **cerr** — Variable

An alternative output stream for errors, analogous to the C `stderr`.

For example, this bare-bones C++ version of the traditional “hello” program uses `<<` and `cout`:

```
#include <iostream.h>

int main(int argc, char **argv)
{
  cout << "Well, hi there.\n";
  return 0;
}
```

Casual use of these operators may be seductive, but—other than in writing throwaway code for your own use—it is not necessarily simpler than managing input and output in any other language. For example, robust code should check the state of the input and output streams between operations (for example, using the method `good`). See Section 3.1.1 “Checking the state of a stream,” page 5. You may also need to adjust maximum input or output field widths, using manipulators like `setw` or `setprecision`.

`<<` Operator on ostream

Write output to an open output stream of class `ostream`. Defined by this library on any *object* of a C++ primitive type, and on other classes of the library. You can overload the definition for any of your own applications' classes.

Returns a reference to the implied argument `*this` (the open stream it writes on), permitting statements like

```
cout << "The value of i is " << i << "\n";
```

`>>` Operator on istream

Read input from an open input stream of class `istream`. Defined by this library on primitive numeric, pointer, and string types; you can extend the definition for any of your own applications' classes.

Returns a reference to the implied argument `*this` (the open stream it reads), permitting multiple inputs in one statement.

3 Stream Classes

The previous chapter referred in passing to the classes `ostream` and `istream`, for output and input respectively. These classes share certain properties, captured in their base class `ios`.

3.1 Shared properties: class `ios`

The base class `ios` provides methods to test and manage the state of input or output streams.

`ios` delegates the job of actually reading and writing bytes to the abstract class `streambuf`, which is designed to provide buffered streams (compatible with C, in the GNU implementation). See Chapter 5 "Using the `streambuf` layer," page 25, for information on the facilities available at the `streambuf` level.

ios::ios ([`streambuf*` *sb* [, `ostream*` *tie*]) Constructor

The `ios` constructor by default initializes a new `ios`, and if you supply a `streambuf` *sb* to associate with it, sets the state `good` in the new `ios` object. It also sets the default properties of the new object.

You can also supply an optional second argument *tie* to the constructor: if present, it is an initial value for `ios::tie`, to associate the new `ios` object with another stream.

ios::~ios () Destructor

The `ios` destructor is virtual, permitting application-specific behavior when a stream is closed—typically, the destructor frees any storage associated with the stream and releases any other associated objects.

3.1.1 Checking the state of a stream

Use this collection of methods to test for (or signal) errors and other exceptional conditions of streams:

`ios::operator void*` () `const` Method

You can do a quick check on the state of the most recent operation on a stream by examining a pointer to the stream itself. The pointer is arbitrary except for its truth value; it is true if no failures have occurred (`ios::fail` is not true). For example, you might ask for input on `cin` only if all prior output operations succeeded:

```
if (cout)
{
  // Everything OK so far
  cin >> new_value;
  ...
}
```

`ios::operator ! () const` Method

In case it is more convenient to check whether something has failed, the operator `!` returns true if `ios::fail` is true (an operation has failed). For example, you might issue an error message if input failed:

```
if (!cin)
{
  // Oops
  cerr << "Eh?\n";
}
```

`iostate` **ios::rdstate** `() const` Method

Return the state flags for this stream. The value is from the enumeration `iostate`. You can test for any combination of

`goodbit` There are no indications of exceptional states on this stream.

`eofbit` End of file.

`failbit` An operation has failed on this stream; this usually indicates bad format of input.

`badbit` The stream is unusable.

`void` **ios::setstate** `(iostate state)` Method

Set the state flag for this stream to *state in addition to* any state flags already set. Synonym (for upward compatibility): `ios::set`.

See `ios::clear` to set the stream state without regard to existing state flags. See `ios::good`, `ios::eof`, `ios::fail`, and `ios::bad`, to test the state.

`int` **ios::good** `() const` Method

Test the state flags associated with this stream; true if no error indicators are set.

`int` **ios::bad** `() const` Method

Test whether a stream is marked as unusable. (Whether `ios::badbit` is set.)

`int` **ios::eof** `()const` Method

True if end of file was reached on this stream. (If `ios::eofbit` is set.)

`int` **ios::fail** `()const` Method

Test for any kind of failure on this stream: *either* some operation failed, *or* the stream is marked as bad. (If either `ios::failbit` or `ios::badbit` is set.)

`void` **ios::clear** `(iostate` *`state`*`)` Method

Set the state indication for this stream to the argument *`state`*. You may call `ios::clear` with no argument, in which case the state is set to `good` (no errors pending).

See `ios::good`, `ios::eof`, `ios::fail`, and `ios::bad`, to test the state; see `ios::set` or `ios::setstate` for an alternative way of setting the state.

3.1.2 Choices in formatting

These methods control (or report on) settings for some details of controlling streams, primarily to do with formatting output:

`char` **ios::fill** `()const` Method

Report on the padding character in use.

`char` **ios::fill** `(char` *`padding`*`)` Method

Set the padding character. You can also use the manipulator `setfill`. See Section 3.1.3 “Changing stream properties in expressions,” page 10.

Default: blank.

`int` **ios::precision** `()const` Method

Report the number of significant digits currently in use for output of floating point numbers.

Default: `6`.

`int` **ios::precision** `(int` *`signif`*`)` Method

Set the number of significant digits (for input and output numeric conversions) to *`signif`*.

You can also use the manipulator `setprecision` for this purpose. See Section 3.1.3 “Changing stream properties using manipulators,” page 10.

`int` **ios::width** `()const` Method

Report the current output field width setting (the number of characters to write on the next ‘`<<`’ output operation).

Default: `0`, which means to use as many characters as necessary.

`int` **ios::width** `(int` *num*`)` Method

Set the input field width setting to *num*. Return the *previous* value for this stream.

This value resets to zero (the default) every time you use '`<<`'; it is essentially an additional implicit argument to that operator. You can also use the manipulator `setw` for this purpose. See Section 3.1.3 "Changing stream properties using manipulators," page 10.

`fmtflags` **ios::flags** `()const` Method

Return the current value of the complete collection of flags controlling the format state. These are the flags and their meanings when set:

`ios::dec`
`ios::oct`
`ios::hex`
: What numeric base to use in converting integers from internal to display representation, or vice versa: decimal, octal, or hexadecimal, respectively. (You can change the base using the manipulator `setbase`, or any of the manipulators `dec`, `oct`, or `hex`; see Section 3.1.3 "Changing stream properties in expressions," page 10.)

: On input, if none of these flags is set, read numeric constants according to the prefix: decimal if no prefix (or a '`.`' suffix), octal if a '`0`' prefix is present, hexadecimal if a '`0x`' prefix is present.

: Default: `dec`.

`ios::fixed`
: Avoid scientific notation, and always show a fixed number of digits after the decimal point, according to the output precision in effect. Use `ios::precision` to set precision.

`ios::left`
`ios::right`
`ios::internal`
: Where output is to appear in a fixed-width field; left-justified, right-justified, or with padding in the middle (e.g. between a numeric sign and the associated value), respectively.

`ios::scientific`
: Use scientific (exponential) notation to display numbers.

`ios::showbase`
: Display the conventional prefix as a visual indicator of the conversion base: no prefix for decimal, '`0`' for octal, '`0x`' for hexadecimal.

`ios::showpoint`
: Display a decimal point and trailing zeros after it to fill out numeric fields, even when redundant.

`ios::showpos`
: Display a positive sign on display of positive numbers.

`ios::skipws`
: Skip white space. (On by default).

`ios::stdio`
: Flush the C `stdio` streams `stdout` and `stderr` after each output operation (for programs that mix C and C++ output conventions).

`ios::unitbuf`
: Flush after each output operation.

`ios::uppercase`
: Use upper-case characters for the non-numeral elements in numeric displays; for instance, '`0X7A`' rather than '`0x7a`', or '`3.14E+09`' rather than '`3.14e+09`'.

`fmtflags` **ios::flags** `(fmtflags` *`value`*`)` — Method

Set *`value`* as the complete collection of flags controlling the format state. The flag values are described under '`ios::flags ()`'.

Use `ios::setf` or `ios::unsetf` to change one property at a time.

`fmtflags` **ios::setf** `(fmtflags` *`flag`*`)` — Method

Set one particular flag (of those described for '`ios::flags ()`'; return the complete collection of flags *previously* in effect. (Use `ios::unsetf` to cancel.)

`fmtflags` **ios::setf** `(fmtflags` *`flag`*`, fmtflags` *`mask`*`)` — Method

Clear the flag values indicated by *`mask`*, then set any of them that are also in *`flag`*. (Flag values are described for

'`ios::flags ()`'.) Return the complete collection of flags *previously* in effect. (See `ios::unsetf` for another way of clearing flags.)

`fmtflags` **ios::unsetf** (`fmtflags` *`flag`*) Method

Make certain *`flag`* (a combination of flag values described for '`ios::flags ()`') is not set for this stream; converse of `ios::setf`. Returns the old values of those flags.

3.1.3 Changing stream properties using manipulators

For convenience, *`manipulators`* provide a way to change certain properties of streams, or otherwise affect them, in the middle of expressions involving '`<<`' or '`>>`'. For example, you might write

```
cout << "|" << setfill('*') << setw(5) << 234 << "|";
```

to produce '`|**234|`' as output.

ws Manipulator

Skip whitespace.

flush Manipulator

Flush an output stream. For example, '`cout << ... <<flush;`' has the same effect as '`cout << ...; cout.flush();`'.

endl Manipulator

Write an end of line character '`\n`', then flushes the output stream.

ends Manipulator

Write '`\0`' (the string terminator character).

setprecision (`int` *`signif`*) Manipulator

You can change the value of `ios::precision` in '`<<`' expressions with the manipulator '`setprecision(`*`signif`*`)`'; for example,

```
cout << setprecision(2) << 4.567;
```

prints '`4.6`'. Requires '`#include <iomanip.h>`'.

setw (`int` *`n`*) Manipulator

You can change the value of `ios::width` in '`<<`' expressions with the manipulator '`setw(`*`n`*`)`'; for example,

```
cout << setw(5) << 234;
```

prints '` 234`' with two leading blanks. Requires '`#include <iomanip.h>`'.

setbase (int *base*) — Manipulator

Where *base* is one of `10` (decimal), `8` (octal), or `16` (hexadecimal), change the base value for numeric representations. Requires '`#include <iomanip.h>`'.

dec — Manipulator

Select decimal base; equivalent to '`setbase(10)`'.

hex — Manipulator

Select hexadecimal base; equivalent to '`setbase(16)`'.

oct — Manipulator

Select octal base; equivalent to '`setbase(8)`'.

setfill (char *padding*) — Manipulator

Set the padding character, in the same way as `ios::fill`. Requires '`#include <iomanip.h>`'.

3.1.4 Extended data fields

A related collection of methods allows you to extend this collection of flags and parameters for your own applications, without risk of conflict between them:

static fmtflags **ios::bitalloc** () — Method

Reserve a bit (the single bit on in the result) to use as a flag. Using `bitalloc` guards against conflict between two packages that use `ios` objects for different purposes.

This method is available for upward compatibility, but is not in the ANSI working paper. The number of bits available is limited; a return value of `0` means no bit is available.

static int **ios::xalloc** () — Method

Reserve space for a long integer or pointer parameter. The result is a unique nonnegative integer. You can use it as an index to `ios::iword` or `ios::pword`. Use `xalloc` to arrange for arbitrary special-purpose data in your `ios` objects, without risk of conflict between packages designed for different purposes.

long& **ios::iword** (int *index*) — Method

Return a reference to arbitrary data, of long integer type, stored in an `ios` instance. *index*, conventionally returned from `ios::xalloc`, identifies what particular data you need.

long **ios::iword** (int *index*) const — Method

Return the actual value of a long integer stored in an `ios`.

`void*&` **ios::pword** `(int index)` Method

Return a reference to an arbitrary pointer, stored in an `ios` instance. *index*, originally returned from `ios::xalloc`, identifies what particular pointer you need.

`void*` **ios::pword** `(int index) const` Method

Return the actual value of a pointer stored in an `ios`.

3.1.5 Synchronizing related streams

You can use these methods to synchronize related streams with one another:

`ostream*` **ios::tie** `() const` Method

Report on what output stream, if any, is to be flushed before accessing this one. A pointer value of `0` means no stream is tied.

`ostream*` **ios::tie** `(ostream* assoc)` Method

Declare that output stream *assoc* must be flushed before accessing this stream.

`int` **ios::sync_with_stdio** `([int switch])` Method

Unless iostreams and C `stdio` are designed to work together, you may have to choose between efficient C++ streams output and output compatible with C `stdio`. Use '`ios::sync_with_stdio()`' to select C compatibility.

The argument *switch* is a GNU extension; use `0` as the argument to choose output that is not necessarily compatible with C `stdio`. The default value for *switch* is `1`.

If you install the `stdio` implementation that comes with GNU `libio`, there are compatible input/output facilities for both C and C++. In that situation, this method is unnecessary—but you may still want to write programs that call it, for portability.

3.1.6 Reaching the underlying `streambuf`

Finally, you can use this method to access the underlying object:

`streambuf*` **ios::rdbuf** `() const` Method

Return a pointer to the `streambuf` object that underlies this `ios`.

3.2 Managing output streams: class `ostream`

Objects of class `ostream` inherit the generic methods from `ios`, and in addition have the following methods available. Declarations for this class come from '`iostream.h`'.

ostream::ostream `()` — Constructor

The simplest form of the constructor for an `ostream` simply allocates a new `ios` object.

ostream::ostream `(streambuf*` *`sb`* `[, ostream` *`tie`*`])` — Constructor

This alternative constructor requires a first argument *`sb`* of type `streambuf*`, to use an existing open stream for output. It also accepts an optional second argument *`tie`*, to specify a related `ostream*` as the initial value for `ios::tie`.

If you give the `ostream` a `streambuf` explicitly, using this constructor, the *`sb`* is *not* destroyed (or deleted or closed) when the `ostream` is destroyed.

3.2.1 Writing on an `ostream`

These methods write on an `ostream` (you may also use the operator `<<`; see Chapter 2 "Operators and Default Streams," page 3).

`ostream&` **ostream::put** `(char` *`c`*`)` — Method

Write the single character *`c`*.

`ostream&` **ostream::write** `(`*`string`*`, int` *`length`*`)` — Method

Write *`length`* characters of a string to this `ostream`, beginning at the pointer *`string`*.

`string` may have any of these types: `char*`, `unsigned char*`, `signed char*`.

`ostream&` **ostream::form** `(const char *`*`format`*`, ...)` — Method

A GNU extension, similar to `fprintf(`*`file`*`,` *`format`*`, ...)`.

`format` is a `printf`-style format control string, which is used to format the (variable number of) arguments, printing the result on this `ostream`. See `ostream::vform` for a version that uses an argument list rather than a variable number of arguments.

`ostream&` **ostream::vform** `(const char *`*`format`*`, va_list` *`args`*`)` — Method

A GNU extension, similar to `vfprintf(`*`file`*`,` *`format`*`,` *`args`*`)`.

format is a `printf`-style format control string, which is used to format the argument list *args*, printing the result on this `ostream`. See `ostream::form` for a version that uses a variable number of arguments rather than an argument list.

3.2.2 Repositioning an `ostream`

You can control the output position (on output streams that actually support positions, typically files) with these methods:

`streampos` **ostream::tellp** `()` Method

Return the current write position in the stream.

`ostream&` **ostream::seekp** `(streampos` *loc*`)` Method

Reset the output position to *loc* (which is usually the result of a previous call to `ostream::tellp`). *loc* specifies an absolute position in the output stream.

`ostream&` **ostream::seekp** `(streamoff` *loc*, *rel*`)` Method

Reset the output position to *loc*, relative to the beginning, end, or current output position in the stream, as indicated by *rel* (a value from the enumeration `ios::seekdir`):

`beg` Interpret *loc* as an absolute offset from the beginning of the file.

`cur` Interpret *loc* as an offset relative to the current output position.

`end` Interpret *loc* as an offset from the current end of the output stream.

3.2.3 Miscellaneous `ostream` utilities

You may need to use these `ostream` methods for housekeeping:

`ostream&` **flush** `()` Method

Deliver any pending buffered output for this `ostream`.

`int` **ostream::opfx** `()` Method

`opfx` is a *prefix* method for operations on `ostream` objects; it is designed to be called before any further processing. See `ostream::osfx` for the converse.

`opfx` tests that the stream is in state `good`, and if so flushes any stream tied to this one.

The result is `1` when `opfx` succeeds; else (if the stream state is not `good`), the result is `0`.

void **ostream::osfx** () — Method

osfx is a *suffix* method for operations on ostream objects; it is designed to be called at the conclusion of any processing. All the ostream methods end by calling osfx. See ostream::opfx for the converse.

If the unitbuf flag is set for this stream, osfx flushes any buffered output for it.

If the stdio flag is set for this stream, osfx flushes any output buffered for the C output streams 'stdout' and 'stderr'.

3.3 Managing input streams: class istream

Class istream objects are specialized for input; as for ostream, they are derived from ios, so you can use any of the general-purpose methods from that base class. Declarations for this class also come from 'iostream.h'.

istream::istream () — Constructor

When used without arguments, the istream constructor simply allocates a new ios object and initializes the input counter (the value reported by istream::gcount) to 0.

istream::istream (streambuf **sb* [, ostream *tie*]) — Constructor

You can also call the constructor with one or two arguments. The first argument *sb* is a streambuf*; if you supply this pointer, the constructor uses that streambuf for input. You can use the second optional argument *tie* to specify a related output stream as the initial value for ios::tie.

If you give the istream a streambuf explicitly, using this constructor, the *sb* is *not* destroyed (or deleted or closed) when the ostream is destroyed.

3.3.1 Reading one character

Use these methods to read a single character from the input stream:

int **istream::get** () — Method

Read a single character (or EOF) from the input stream, returning it (coerced to an unsigned char) as the result.

istream& **istream::get** (char& *c*) — Method

Read a single character from the input stream, into &*c*.

int **istream::peek** () Method

Return the next available input character, but *without* changing the current input position.

3.3.2 Reading strings

Use these methods to read strings (for example, a line at a time) from the input stream:

istream& **istream::get** (char* *c*, int *len* **[**, char *delim***]**) Method

Read a string from the input stream, into the array at *c*.

The remaining arguments limit how much to read: up to 'len-1' characters, or up to (but not including) the first occurrence in the input of a particular delimiter character *delim*—newline (\n) by default. (Naturally, if the stream reaches end of file first, that too will terminate reading.)

If *delim* was present in the input, it remains available as if unread; to discard it instead, see iostream::getline.

get writes '\0' at the end of the string, regardless of which condition terminates the read.

istream& **istream::get** (streambuf& *sb* **[**, char *delim***]**) Method

Read characters from the input stream and copy them on the streambuf object *sb*. Copying ends either just before the next instance of the delimiter character *delim* (newline \n by default), or when either stream ends. If *delim* was present in the input, it remains available as if unread.

istream& **istream::getline** (*charptr*, int *len* **[**, char *delim***]**) Method

Read a line from the input stream, into the array at *charptr*. *charptr* may be any of three kinds of pointer: char*, unsigned char*, or signed char*.

The remaining arguments limit how much to read: up to (but not including) the first occurrence in the input of a line delimiter character *delim*—newline (\n) by default, or up to 'len-1' characters (or to end of file, if that happens sooner).

If getline succeeds in reading a "full line", it also discards the trailing delimiter character from the input stream. (To preserve it as available input, see the similar form of iostream::get.)

If *delim* was *not* found before *len* characters or end of file, `getline` sets the `ios::fail` flag, as well as the `ios::eof` flag if appropriate.

`getline` writes a null character at the end of the string, regardless of which condition terminates the read.

`istream&` **istream::read** (*pointer*, int *len*) Method

Read *len* bytes into the location at *pointer*, unless the input ends first.

pointer may be of type `char*`, `void*`, `unsigned char*`, or `signed char*`.

If the `istream` ends before reading *len* bytes, `read` sets the `ios::fail` flag.

`istream&` **istream::gets** (char `**`*s* [, char *delim*]) Method

A GNU extension, to read an arbitrarily long string from the current input position to the next instance of the *delim* character (newline `\n` by default).

To permit reading a string of arbitrary length, `gets` allocates whatever memory is required. Notice that the first argument *s* is an address to record a character pointer, rather than the pointer itself.

`istream&` **istream::scan** (const char *format ...) Method

A GNU extension, similar to `fscanf(`*file*`,` *format*`, ...)`. The *format* is a `scanf`-style format control string, which is used to read the variables in the remainder of the argument list from the `istream`.

`istream&` **istream::vscan** (const char *format, va_list args) Method

Like `istream::scan`, but takes a single `va_list` argument.

3.3.3 Repositioning an `istream`

Use these methods to control the current input position:

`streampos` **istream::tellg** () Method

Return the current read position, so that you can save it and return to it later with `istream::seekg`.

`istream&` **istream::seekg** (streampos *p*) Method

Reset the input pointer (if the input device permits it) to *p*, usually the result of an earlier call to `istream::tellg`.

`istream&` **istream::seekg** (`streamoff` *offset*, `ios::seek_dir` *ref*) — Method

Reset the input pointer (if the input device permits it) to *offset* characters from the beginning of the input, the current position, or the end of input. Specify how to interpret *offset* with one of these values for the second argument:

`ios::beg` Interpret *loc* as an absolute offset from the beginning of the file.

`ios::cur` Interpret *loc* as an offset relative to the current output position.

`ios::end` Interpret *loc* as an offset from the current end of the output stream.

3.3.4 Miscellaneous `istream` utilities

Use these methods for housekeeping on `istream` objects:

`int` **istream::gcount** () — Method

Report how many characters were read from this `istream` in the last unformatted input operation.

`int` **istream::ipfx** (`int` *keepwhite*) — Method

Ensure that the `istream` object is ready for reading; check for errors and end of file and flush any tied stream. `ipfx` skips whitespace if you specify `0` as the *keepwhite* argument, *and* `ios::skipws` is set for this stream.

To avoid skipping whitespace (regardless of the `skipws` setting on the stream), use `1` as the argument.

Call `istream::ipfx` to simplify writing your own methods for reading `istream` objects.

`void` **istream::isfx** () — Method

A placeholder for compliance with the draft ANSI standard; this method does nothing whatever.

If you wish to write portable standard-conforming code on `istream` objects, call `isfx` after any operation that reads from an `istream`; if `istream::ipfx` has any special effects that must be cancelled when done, `istream::isfx` will cancel them.

`istream&` **istream::ignore** (**[**`int` *n***]** **[**, `int` *delim***]**) — Method

Discard some number of characters pending input. The first optional argument *n* specifies how many characters to skip.

The second optional argument `delim` specifies a “boundary” character: `ignore` returns immediately if this character appears in the input.

By default, `delim` is `EOF`; that is, if you do not specify a second argument, only the count *n* restricts how much to ignore (while input is still available).

If you do not specify how many characters to ignore, `ignore` returns after discarding only one character.

`istream&` **istream::putback** `(char ch)` Method

Attempts to back up one character, replacing the character backed-up over by *ch*. Returns `EOF` if this is not allowed. Putting back the most recently read character is always allowed. (This method corresponds to the C function `ungetc`.)

`istream&` **istream::unget** `()` Method

Attempt to back up one character.

3.4 Input and output together: class `iostream`

If you need to use the same stream for input and output, you can use an object of the class `iostream`, which is derived from *both* `istream` and `ostream`.

The constructors for `iostream` behave just like the constructors for `istream`.

iostream::iostream `()` Constructor

When used without arguments, the `iostream` constructor simply allocates a new `ios` object, and initializes the input counter (the value reported by `istream::gcount`) to `0`.

iostream::iostream `(streambuf* sb [, ostream* tie])` Constructor

You can also call a constructor with one or two arguments. The first argument *sb* is a `streambuf*`; if you supply this pointer, the constructor uses that `streambuf` for input and output.

You can use the optional second argument *tie* (an `ostream*`) to specify a related output stream as the initial value for `ios::tie`.

As for `ostream` and `istream`, `iostream` simply uses the `ios` destructor. However, an `iostream` is not deleted by its destructor.

You can use all the `istream`, `ostream`, and `ios` methods with an `iostream` object.

4 Classes for Files and Strings

There are two very common special cases of input and output: using files, and using strings in memory.

`libio` defines four specialized classes for these cases:

`ifstream` Methods for reading files.

`ofstream` Methods for writing files.

`istrstream`
Methods for reading strings from memory.

`ostrstream`
Methods for writing strings in memory.

4.1 Reading and writing files

These methods are declared in '`fstream.h`'.

You can read data from class `ifstream` with any operation from class `istream`. There are also a few specialized facilities:

ifstream::ifstream `()` — Constructor

Make an `ifstream` associated with a new file for input. (If you use this version of the constructor, you need to call `ifstream::open` before actually reading anything)

ifstream::ifstream `(int` *`fd`*`)` — Constructor

Make an `ifstream` for reading from a file that was already open, using file descriptor *`fd`*. (This constructor is compatible with other versions of iostreams for POSIX systems, but is not part of the ANSI working paper.)

ifstream::ifstream `(const char*` *`fname`* **[**`, int` *`mode`* **[**`, int` *`prot`***]]**`)` — Constructor

Open a file `*`*`fname`* for this `ifstream` object.

By default, the file is opened for input (with `ios::in` as *`mode`*). If you use this constructor, the file will be closed when the `ifstream` is destroyed.

You can use the optional argument *`mode`* to specify how to open the file, by combining these enumerated values (with '`|`' bitwise or). (These values are actually defined in class `ios`, so that all file-related streams may inherit them.) Only some of these modes are defined in the latest draft ANSI specification; if portability is important, you may wish to avoid the others.

`ios::in` Open for input. (Included in ANSI draft.)

`ios::out` Open for output. (Included in ANSI draft.)

`ios::ate` Set the initial input (or output) position to the end of the file.

`ios::app` Seek to end of file before each write. (Included in ANSI draft.)

`ios::trunc`
Guarantee a fresh file; discard any contents that were previously associated with it.

`ios::nocreate`
Guarantee an existing file; fail if the specified file did not already exist.

`ios::noreplace`
Guarantee a new file; fail if the specified file already existed.

`ios::bin` Open as a binary file (on systems where binary and text files have different properties, typically how '`\n`' is mapped; included in ANSI draft).

The last optional argument *prot* is specific to Unix-like systems; it specifies the file protection (by default '`644`').

void **ifstream::open** (const char* *fname* [, int *mode* [, int *prot*]]) — Method

Open a file explicitly after the associated `ifstream` object already exists (for instance, after using the default constructor). The arguments, options and defaults all have the same meanings as in the fully specified `ifstream` constructor.

You can write data to class `ofstream` with any operation from class `ostream`. There are also a few specialized facilities:

ofstream::ofstream () — Constructor

Make an `ofstream` associated with a new file for output.

ofstream::ofstream (int *fd*) — Constructor

Make an `ofstream` for writing to a file that was already open, using file descriptor *fd*.

ofstream::ofstream (const char* *fname* [, int *mode* [, int *prot*]]) — Constructor

Open a file `*`*fname* for this `ofstream` object.

By default, the file is opened for output (with `ios::out` as *mode*). You can use the optional argument *mode* to specify how to open the file, just as described for `ifstream::ifstream`.

The last optional argument *prot* specifies the file protection (by default '`644`').

ofstream::~ofstream `()` — Destructor

The files associated with `ofstream` objects are closed when the corresponding object is destroyed.

`void` **ofstream::open** `(const char* fname [, int mode [, int prot]])` — Method

Open a file explicitly after the associated `ofstream` object already exists (for instance, after using the default constructor). The arguments, options and defaults all have the same meanings as in the fully specified `ofstream` constructor.

The class `fstream` combines the facilities of `ifstream` and `ofstream`, just as `iostream` combines `istream` and `ostream`.

The class `fstreambase` underlies both `ifstream` and `ofstream`. They both inherit this additional method:

`void` **fstreambase::close** `()` — Method

Close the file associated with this object, and set `ios::fail` in this object to mark the event.

4.2 Reading and writing in memory

The classes `istrstream`, `ostrstream`, and `strstream` provide some additional features for reading and writing strings in memory—both static strings, and dynamically allocated strings. The underlying class `strstreambase` provides some features common to all three; `strstreambuf` underlies that in turn.

istrstream::istrstream `(const char* str [, int size])` — Constructor

Associate the new input string class `istrstream` with an existing static string starting at *str*, of size *size*. If you do not specify *size*, the string is treated as a `NUL` terminated string.

ostrstream::ostrstream `()` — Constructor

Create a new stream for output to a dynamically managed string, which will grow as needed.

ostrstream::ostrstream (char* *str*, int *size* [, int *mode*]) — Constructor

A new stream for output to a statically defined string of length *size*, starting at *str*. You may optionally specify one of the modes described for `ifstream::ifstream`; if you do not specify one, the new stream is simply open for output, with mode `ios::out`.

int **ostrstream::pcount** () — Method

Report the current length of the string associated with this `ostrstream`.

char* **ostrstream::str** () — Method

A pointer to the string managed by this `ostrstream`. Implies '`ostrstream::freeze()`'.

Note that if you want the string to be nul-terminated, you must do that yourself (perhaps by writing ends to the stream).

void **ostrstream::freeze** ([int *n*]) — Method

If *n* is nonzero (the default), declare that the string associated with this `ostrstream` is not to change dynamically; while frozen, it will not be reallocated if it needs more space, and it will not be deallocated when the `ostrstream` is destroyed. Use '`freeze(1)`' if you refer to the string as a pointer after creating it via `ostrstream` facilities.

'`freeze(0)`' cancels this declaration, allowing a dynamically allocated string to be freed when its `ostrstream` is destroyed.

If this `ostrstream` is already static—that is, if it was created to manage an existing statically allocated string—`freeze` is unnecessary, and has no effect.

int **ostrstream::frozen** () — Method

Test whether `freeze(1)` is in effect for this string.

strstreambuf* **strstreambase::rdbuf** () — Method

A pointer to the underlying `strstreambuf`.

5 Using the `streambuf` Layer

The `istream` and `ostream` classes are meant to handle conversion between objects in your program and their textual representation.

By contrast, the underlying `streambuf` class is for transferring raw bytes between your program, and input sources or output sinks. Different `streambuf` subclasses connect to different kinds of sources and sinks.

The GNU implementation of `streambuf` is still evolving; we describe only some of the highlights.

5.1 Areas of a `streambuf`

Streambuf buffer management is fairly sophisticated (this is a nice way to say "complicated"). The standard protocol has the following "areas":

- The *put area* contains characters waiting for output.
- The *get area* contains characters available for reading.

The GNU `streambuf` design extends this, but the details are still evolving.

The following methods are used to manipulate these areas. These are all protected methods, which are intended to be used by virtual function in classes derived from `streambuf`. They are also all ANSI/ISO-standard, and the ugly names are traditional. (Note that if a pointer points to the 'end' of an area, it means that it points to the character after the area.)

`char*` **streambuf::pbase** `()const` Method

Returns a pointer to the start of the put area.

`char*` **streambuf::epptr** `()const` Method

Returns a pointer to the end of the put area.

`char*` **streambuf::pptr** `()const` Method

If `pptr() < epptr ()`, the `pptr()` returns a pointer to the current put position. (In that case, the next write will overwrite `*pptr()`, and increment `pptr()`.) Otherwise, there is no put position available (and the next character written will cause `streambuf::overflow` to be called).

`void` **streambuf::pbump** `(int N)` Method

Add *N* to the current put pointer. No error checking is done.

void **streambuf::setp** (char* *P*, char* *E*) Method

Sets the start of the put area to *P*, the end of the put area to *E*, and the current put pointer to *P* (also).

char* **streambuf::eback** ()const Method

Returns a pointer to the start of the get area.

char* **streambuf::egptr** ()const Method

Returns a pointer to the end of the get area.

char* **streambuf::gptr** ()const Method

If `gptr() < egptr ()`, then `gptr()` returns a pointer to the current get position. (In that case the next read will read `*gptr()`, and possibly increment `gptr()`.) Otherwise, there is no read position available (and the next read will cause `streambuf::underflow` to be called).

void **streambuf:gbump** (int *N*) Method

Add *N* to the current get pointer. No error checking is done.

void **streambuf::setg** (char* *B*, char* *P*, char* *E*) Method

Sets the start of the get area to *B*, the end of the get area to *E*, and the current put pointer to *P*.

5.2 Simple output re-direction by redefining `overflow`

Suppose you have a function `write_to_window` that writes characters to a `window` object. If you want to use the ostream function to write to it, here is one (portable) way to do it. This depends on the default buffering (if any).

```
#include <iostream.h>
/* Returns number of characters successfully written to win. */
extern int write_to_window (window* win, char* text, int length);

class windowbuf : public streambuf {
    window* win;
  public:
    windowbuf (window* w) { win = w; }
    int sync ();
    int overflow (int ch);
    // Defining xsputn is an optional optimization.
    // (streamsize was recently added to ANSI C++, not portable yet.)
    streamsize xsputn (char* text, streamsize n);
};

int windowbuf::sync ()
{ streamsize n = pptr () - pbase ();
  return (n && write_to_window (win, pbase (), n) != n) ? EOF : 0;
}

int windowbuf::overflow (int ch)
{ streamsize n = pptr () - pbase ();
  if (n && sync ())
    return EOF;
  if (ch != EOF)
    {
      char cbuf[1];
      cbuf[0] = ch;
      if (write_to_window (win, cbuf, 1) != 1)
        return EOF;
    }
  pbump (-n);  // Reset pptr().
  return 0;
}

streamsize windowbuf::xsputn (char* text, streamsize n)
{ return sync () == EOF ? 0 : write_to_window (win, text, n); }

int
main (int argc, char**argv)
{
  window *win = ...;
  windowbuf wbuf(win);
  ostream wstr(&wbuf);
  wstr << "Hello world!\n";
}
```

5.3 C-style formatting for `streambuf` objects

The GNU `streambuf` class supports `printf`-like formatting and scanning.

int **streambuf::vform** (const char **format*, ...) — Method

Similar to `fprintf(`*file*`, `*format*`, ...)`. The *format* is a `printf`-style format control string, which is used to format the (variable number of) arguments, printing the result on the `this` streambuf. The result is the number of characters printed.

int **streambuf::vform** (const char **format*, va_list *args*) — Method

Similar to `vfprintf(`*file*`, `*format*`, `*args*`)`. The *format* is a `printf`-style format control string, which is used to format the argument list *args*, printing the result on the `this` streambuf. The result is the number of characters printed.

int **streambuf::scan** (const char **format*, ...) — Method

Similar to `fscanf(`*file*`, `*format*`, ...)`. The *format* is a `scanf`-style format control string, which is used to read the (variable number of) arguments from the `this` streambuf. The result is the number of items assigned, or `EOF` in case of input failure before any conversion.

int **streambuf::vscan** (const char **format*, va_list *args*) — Method

Like `streambuf::scan`, but takes a single `va_list` argument.

5.4 Wrappers for C `stdio`

A *stdiobuf* is a `streambuf` object that points to a `FILE` object (as defined by `stdio.h`). All `streambuf` operations on the `stdiobuf` are forwarded to the `FILE`. Thus the `stdiobuf` object provides a wrapper around a `FILE`, allowing use of `streambuf` operations on a `FILE`. This can be useful when mixing C code with C++ code.

The pre-defined streams `cin`, `cout`, and `cerr` are normally implemented as `stdiobuf` objects that point to respectively `stdin`, `stdout`, and `stderr`. This is convenient, but it does cost some extra overhead.

If you set things up to use the implementation of `stdio` provided with this library, then `cin`, `cout`, and `cerr` will be set up to to use `stdiobuf`

objects, since you get their benefits for free. See Chapter 6 "C Input and Output," page 33.

5.5 Reading/writing from/to a pipe

The *procbuf* class is a GNU extension. It is derived from `streambuf`. A `procbuf` can be *closed* (in which case it does nothing), or *open* (in which case it allows communicating through a pipe with some other program).

procbuf::procbuf `()` — Constructor

Creates a `procbuf` in a *closed* state.

`procbuf*` **procbuf::open** `(const char *command, int mode)` — Method

Uses the shell ('`/bin/sh`') to run a program specified by *command*.

If *mode* is '`ios::in`', standard output from the program is sent to a pipe; you can read from the pipe by reading from the `procbuf`. (This is similar to '`popen(command, "r")`'.)

If *mode* is '`ios::out`', output written written to the `procbuf` is written to a pipe; the program is set up to read its standard input from (the other end of) the pipe. (This is similar to '`popen(command, "w")`'.)

The `procbuf` must start out in the *closed* state. Returns '`*this`' on success, and '`NULL`' on failure.

procbuf::procbuf `(const char *command, int mode)` — Constructor

Calls '`procbuf::open (command, mode)`'.

`procbuf*` **procbuf::close** `()` — Method

Waits for the program to finish executing, and then cleans up the resources used. Returns '`*this`' on success, and '`NULL`' on failure.

procbuf::~procbuf `()` — Destructor

Calls '`procbuf::close`'.

5.6 Backing up

The GNU iostream library allows you to ask a `streambuf` to remember the current position. This allows you to go back to this position later, after reading further. You can back up arbitrary amounts, even on unbuffered files or multiple buffers' worth, as long as you tell the library

in advance. This unbounded backup is very useful for scanning and parsing applications. This example shows a typical scenario:

```
// Read either "dog", "hound", or "hounddog".
// If "dog" is found, return 1.
// If "hound" is found, return 2.
// If "hounddog" is found, return 3.
// If none of these are found, return -1.
int my_scan(streambuf* sb)
{
    streammarker fence(sb);
    char buffer[20];
    // Try reading "hounddog":
    if (sb->sgetn(buffer, 8) == 8
        && strncmp(buffer, "hounddog", 8) == 0)
      return 3;
    // No, no "hounddog":  Back up to 'fence'
    sb->seekmark(fence); //
    // ... and try reading "dog":
    if (sb->sgetn(buffer, 3) == 3
        && strncmp(buffer, "dog", 3) == 0)
      return 1;
    // No, no "dog" either:  Back up to 'fence'
    sb->seekmark(fence); //
    // ... and try reading "hound":
    if (sb->sgetn(buffer, 5) == 5
        && strncmp(buffer, "hound", 5) == 0)
      return 2;
    // No, no "hound" either:  Back up and signal failure.
    sb->seekmark(fence); // Backup to 'fence'
    return -1;
}
```

streammarker::streammarker (`streambuf*` *sbuf*) — Constructor

Create a `streammarker` associated with *sbuf* that remembers the current position of the get pointer.

`int` **streammarker::delta** (`streammarker&` *mark2*) — Method

Return the difference between the get positions corresponding to `*this` and *mark2* (which must point into the same `streambuffer` as `this`).

`int` **streammarker::delta** () — Method

Return the position relative to the streambuffer's current get position.

`int` **streambuf::seekmark** (`streammarker&` *`mark`*) — Method

Move the get pointer to where it (logically) was when *`mark`* was constructed.

5.7 Forwarding I/O activity

An *indirectbuf* is one that forwards all of its I/O requests to another streambuf.

An `indirectbuf` can be used to implement Common Lisp synonym-streams and two-way-streams:

```
class synonymbuf : public indirectbuf {
   Symbol *sym;
   synonymbuf(Symbol *s) { sym = s; }
   virtual streambuf *lookup_stream(int mode) {
       return coerce_to_streambuf(lookup_value(sym)); }
};
```

6 C Input and Output

`libio` is distributed with a complete implementation of the ANSI C `stdio` facility. It is implemented using `streambuf` objects. See Section 5.4 "Wrappers for C `stdio`," page 28.

The `stdio` package is intended as a replacement for the whatever `stdio` is in your C library. Since `stdio` works best when you build `libc` to contain it, and that may be inconvenient, it is not installed by default.

Extensions beyond ANSI:

- A stdio `FILE` is identical to a streambuf. Hence there is no need to worry about synchronizing C and C++ input/output—they are by definition always synchronized.
- If you create a new streambuf sub-class (in C++), you can use it as a `FILE` from C. Thus the system is extensible using the standard `streambuf` protocol.
- You can arbitrarily mix reading and writing, without having to seek in between.
- Unbounded `ungetc()` buffer.

Index

(

() 5, 6

>

>>

on istream 4

<

<<

on ostream 4

B

badbit 6
beg 14

C

cerr 3
cin 3
class fstream 23
class fstreambase 23
class ifstream 21
class istrstream 23
class ostream 22
class ostrstream 23
class strstream 23
class strstreambase 23
class strstreambuf 23
cout 3
cur 14

D

dec 11
destructor for iostream 19

E

end 14
endl 10
ends 10
eofbit 6

F

failbit 6
flush 10, 14
fstream 23
fstreambase 23
fstreambase::close 23

G

get area 25
goodbit 6

H

hex 11

I

ifstream 21
ifstream::ifstream 21
ifstream::open 22
ios::~ios 5
ios::app 22
ios::ate 22
ios::bad 6
ios::beg 18
ios::bin 22
ios::bitalloc 11
ios::clear 7
ios::cur 18
ios::dec 8
ios::end 18
ios::eof 7
ios::fail 7
ios::fill 7
ios::fixed 8
ios::flags 8, 9
ios::good 6
ios::hex 8
ios::in 22
ios::internal 8
ios::ios 5
ios::iword 11
ios::left 8
ios::nocreate 22

ios::noreplace 22
ios::oct 8
ios::out 22
ios::precision 7
ios::pword 12
ios::rdbuf 12
ios::rdstate 6
ios::right 8
ios::scientific 9
ios::seekdir 14
ios::set 6
ios::setf 9
ios::setstate 6
ios::showbase 9
ios::showpoint 9
ios::showpos 9
ios::skipws 9
ios::stdio 9
ios::sync_with_stdio 12
ios::tie 12
ios::trunc 22
ios::unitbuf 9
ios::unsetf 10
ios::uppercase 9
ios::width 7, 8
ios::xalloc 11
iostream **destructor** 19
iostream::iostream 19
istream::gcount 18
istream::get 15, 16
istream::getline 16
istream::gets 17
istream::ignore 18
istream::ipfx 18
istream::isfx 18
istream::istream 15
istream::peek 16
istream::putback 19
istream::read 17
istream::scan 17
istream::seekg 17, 18
istream::tellg 17
istream::unget 19
istream::vscan 17
istrstream 21, 23
istrstream::istrstream 23

O

oct 11
ofstream 21
ofstream::~ofstream 23
ofstream::ofstream 22
ofstream::open 23
ostream 22
ostream::form 13
ostream::opfx 14
ostream::osfx 15
ostream::ostream 13
ostream::put 13
ostream::seekp 14
ostream::tellp 14
ostream::vform 13
ostream::write 13
ostrstream 21, 23
ostrstream::freeze 24
ostrstream::frozen 24
ostrstream::ostrstream 23, 24
ostrstream::pcount 24
ostrstream::str 24

P

procbuf::~procbuf 29
procbuf::close 29
procbuf::open 29
procbuf::procbuf 29
put area 25

S

setbase 11
setfill 11
setprecision 7, 10
setting ios::precision 7
setting ios::width 8
setw 8, 10
streambuf::eback 26
streambuf::egptr 26
streambuf::epptr 25
streambuf::gptr 26
streambuf::pbase 25
streambuf::pbump 25
streambuf::pptr 25
streambuf::scan 28
streambuf::seekmark 30
streambuf::setg 26

streambuf::setp 26
streambuf::vform 28
streambuf::vscan 28
streambuf:gbump 26
streammarker::delta 30
streammarker::streammarker 30
strstream 23
strstreambase 23
strstreambase::rdbuf 24
strstreambuf 23

W

ws 10

The Cygnus C Support Library

Full Configuration

`libc` 1.4
May 1993

Steve Chamberlain
Roland Pesch
Cygnus Support

sac@cygnus.com, pesch@cygnus.com *The Cygnus C Support Library*
Copyright © 1992, 1993 Cygnus Support

'libc' includes software developed by the University of California, Berkeley and its contributors.

'libc' includes software developed by Martin Jackson, Graham Haley and Steve Chamberlain of Tadpole Technology and released to Cygnus.

'libc' uses floating point converstion software developed at AT&T, which includes this copyright information:

The author of this software is David M. Gay.

Copyright (c) 1991 by AT&T.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, subject to the terms of the GNU General Public License, which includes the provision 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 manual into another language, under the above conditions for modified versions.

Table of Contents

1 Standard Utility Functions ('stdlib.h') 1

1.1 abort—abnormal termination of a program 2
1.2 abs—integer absolute value (magnitude) 3
1.3 assert—Macro for Debugging Diagnostics 4
1.4 atexit—request execution of functions at program exit .. 5
1.5 atof, atoff—string to double or float 6
1.6 atoi, atol—string to integer 7
1.7 atol—convert string to long 8
1.8 bsearch—binary search 9
1.9 calloc—allocate space for arrays 10
1.10 div—divide two integers 11
1.11 ecvt,ecvtf,fcvt,fcvtf—double or float to string 12
1.12 gvcvt, gcvtf—format double or float as string......... 13
1.13 ecvtbuf, fcvtbuf—double or float to string 14
1.14 exit—end program execution.......................... 15
1.15 getenv—look up environment variable 16
1.16 labs—long integer absolute value 17
1.17 ldiv—divide two long integers......................... 18
1.18 malloc, realloc, free—manage memory.............. 19
1.19 mbtowc—minimal multibyte to wide char converter.... 21
1.20 qsort—sort an array................................. 22
1.21 rand, srand—pseudo-random numbers 23
1.22 strtod, strtodf—string to double or float............. 24
1.23 strtol—string to long 25
1.24 strtoul—string to unsigned long...................... 27
1.25 system—execute command string...................... 29
1.26 wctomb—minimal wide char to multibyte converter.... 30

2 Character Type Macros and Functions ('ctype.h') 31

2.1 isalnum—alphanumeric character predicate............ 32
2.2 isalpha—alphabetic character predicate 33
2.3 isascii—ASCII character predicate.................... 34
2.4 iscntrl—control character predicate 35
2.5 isdigit—decimal digit predicate 36
2.6 islower—lower-case character predicate 37
2.7 isprint, isgraph—printable character predicates...... 38
2.8 ispunct—punctuation character predicate.............. 39
2.9 isspace—whitespace character predicate............... 40
2.10 isupper—uppercase character predicate 41
2.11 isxdigit—hexadecimal digit predicate................ 42

2.12 toascii—force integers to ASCII range 43
2.13 tolower—translate characters to lower case 44
2.14 toupper—translate characters to upper case 45

3 Input and Output ('stdio.h') 47

3.1 clearerr—clear file or stream error indicator 48
3.2 fclose—close a file 49
3.3 feof—test for end of file 50
3.4 ferror—test whether read/write error has occurred 51
3.5 fflush—flush buffered file output 52
3.6 fgetc—get a character from a file or stream 53
3.7 fgetpos—record position in a stream or file 54
3.8 fgets—get character string from a file or stream 55
3.9 fiprintf—format output to file (integer only) 56
3.10 fopen—open a file.................................... 57
3.11 fdopen—turn open file into a stream 59
3.12 fputc—write a character on a stream or file 60
3.13 fputs—write a character string in a file or stream 61
3.14 fread—read array elements from a file 62
3.15 freopen—open a file using an existing file descriptor .. 63
3.16 fseek—set file position 64
3.17 fsetpos—restore position of a stream or file 65
3.18 ftell—return position in a stream or file 66
3.19 fwrite—write array elements 67
3.20 getc—read a character (macro) 68
3.21 getchar—read a character (macro) 69
3.22 gets—get character string (obsolete, use fgets instead) 70
3.23 iprintf—write formatted output (integer only) 71
3.24 mktemp, mkstemp—generate unused file name.......... 72
3.25 perror—print an error message on standard error 73
3.26 putc—write a character (macro) 74
3.27 putchar—write a character (macro) 75
3.28 puts—write a character string......................... 76
3.29 remove—delete a file's name 77
3.30 rename—rename a file 78
3.31 rewind—reinitialize a file or stream 79
3.32 setbuf—specify full buffering for a file or stream...... 80
3.33 setvbuf—specify file or stream buffering 81
3.34 siprintf—write formatted output (integer only) 83
3.35 printf, fprintf, sprintf—format output............. 84
3.36 scanf, fscanf, sscanf—scan and format input........ 88
3.37 tmpfile—create a temporary file 93
3.38 tmpnam, tempnam—name for a temporary file........... 94

3.39 vprintf, vfprintf, vsprintf—format argument list . . 96

4 Strings and Memory ('string.h') 97
4.1 bcmp—compare two memory areas 98
4.2 bcopy—copy memory regions 99
4.3 bzero—initialize memory to zero 100
4.4 index—search for character in string 101
4.5 memchr—find character in memory 102
4.6 memcmp—compare two memory areas 103
4.7 memcpy—copy memory regions 104
4.8 memmove—move possibly overlapping memory 105
4.9 memset—set an area of memory 106
4.10 rindex—reverse search for character in string 107
4.11 strcat—concatenate strings 108
4.12 strchr—search for character in string 109
4.13 strcmp—character string compare 110
4.14 strcoll—locale specific character string compare 111
4.15 strcpy—copy string 112
4.16 strcspn—count chars not in string 113
4.17 strerror—convert error number to string 114
4.18 strlen—character string length 117
4.19 strncat—concatenate strings 118
4.20 strncmp—character string compare 119
4.21 strncpy—counted copy string 120
4.22 strpbrk—find chars in string 121
4.23 strrchr—reverse search for character in string 122
4.24 strspn—find initial match 123
4.25 strstr—find string segment 124
4.26 strtok—get next token from a string 125
4.27 strxfrm—transform string 126

5 Signal Handling ('signal.h') 127
5.1 raise—send a signal 128
5.2 signal—specify handler subroutine for a signal 129

6 Time Functions ('time.h') 131
6.1 asctime—format time as string 132
6.2 clock—cumulative processor time 133
6.3 ctime—convert time to local and format as string 134
6.4 difftime—subtract two times 135
6.5 gmtime—convert time to UTC traditional form 136
6.6 localtime—convert time to local representation 137
6.7 mktime—convert time to arithmetic representation 138
6.8 strftime—flexible calendar time formatter 139

6.9 `time`—get current calendar time (as single number) . . . 141

7 Locale (‘`locale.h`’) . . . 143

7.1 `setlocale`, `localeconv`—select or query locale . . . 146

8 Reentrancy . . . 149

9 System Calls . . . 153

9.1 Definitions for OS interface . . . 153
9.2 Reentrant covers for OS subroutines . . . 158

10 Variable Argument Lists . . . 161

10.1 ANSI-standard macros, ‘`stdarg.h`’ . . . 161
10.1.1 Initialize variable argument list . . . 162
10.1.2 Extract a value from argument list . . . 163
10.1.3 Abandon a variable argument list . . . 164
10.2 Traditional macros, ‘`varargs.h`’ . . . 164
10.2.1 Declare variable arguments . . . 165
10.2.2 Initialize variable argument list . . . 166
10.2.3 Extract a value from argument list . . . 167
10.2.4 Abandon a variable argument list . . . 168

Index . . . 169

1 Standard Utility Functions (‘stdlib.h’)

This chapter groups utility functions useful in a variety of programs. The corresponding declarations are in the header file ‘stdlib.h’.

1.1 abort—abnormal termination of a program

Synopsis

```
#include <stdlib.h>
void abort(void);
```

Description

Use `abort` to signal that your program has detected a condition it cannot deal with. Normally, `abort` ends your program's execution.

Before terminating your program, `abort` raises the exception `SIGABRT` (using '`raise(SIGABRT)`'). If you have used `signal` to register an exception handler for this condition, that handler has the opportunity to retain control, thereby avoiding program termination.

In this implementation, `abort` does not perform any stream- or file-related cleanup (the host environment may do so; if not, you can arrange for your program to do its own cleanup with a `SIGABRT` exception handler).

Returns

`abort` does not return to its caller.

Portability

ANSI C requires `abort`.

Supporting OS subroutines required: `getpid`, `kill`.

1.2 abs—integer absolute value (magnitude)

Synopsis

```
#include <stdlib.h>
int abs(int i);
```

Description

`abs` returns $|x|$, the absolute value of *i* (also called the magnitude of *i*). That is, if *i* is negative, the result is the opposite of *i*, but if *i* is nonnegative the result is *i*.

The similar function `labs` uses and returns `long` rather than `int` values.

Returns

The result is a nonnegative integer.

Portability

`abs` is ANSI.

No supporting OS subroutines are required.

1.3 `assert`—Macro for Debugging Diagnostics

Synopsis

```
#include <assert.h>
#include <stdlib.h>
void assert(int expression);
```

Description

Use this macro to embed debuggging diagnostic statements in your programs. The argument *`expression`* should be an expression which evaluates to true (nonzero) when your program is working as you intended.

When *`expression`* evaluates to false (zero), `assert` calls `abort`, after first printing a message showing what failed and where:

```
Assertion failed: expression, file filename, line lineno
```

The macro is defined to permit you to turn off all uses of `assert` at compile time by defining `NDEBUG` as a preprocessor variable. If you do this, the `assert` macro expands to

```
(void(0))
```

Returns

`assert` does not return a value.

Portability

The `assert` macro is required by ANSI, as is the behavior when `NDEBUG` is defined.

Supporting OS subroutines required (only if enabled): `close`, `fstat`, `getpid`, `isatty`, `kill`, `lseek`, `read`, `sbrk`, `write`.

1.4 `atexit`—request execution of functions at program exit

Synopsis

```
#include <stdlib.h>
int atexit(void (*function)(void);
```

Description

You can use `atexit` to enroll functions in a list of functions that will be called when your program terminates normally. The argument is a pointer to a user-defined function (which must not require arguments and must not return a result).

The functions are kept in a LIFO stack; that is, the last function enrolled by `atexit` will be the first to execute when your program exits.

There is no built-in limit to the number of functions you can enroll in this list; however, after every group of 32 functions is enrolled, `atexit` will call `malloc` to get space for the next part of the list. The initial list of 32 functions is statically allocated, so you can always count on at least that many slots available.

Returns

`atexit` returns `0` if it succeeds in enrolling your function, `-1` if it fails (possible only if no space was available for `malloc` to extend the list of functions).

Portability

`atexit` is required by the ANSI standard, which also specifies that implementations must support enrolling at least 32 functions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.5 `atof`, `atoff`—string to double or float

Synopsis

```
#include <stdlib.h>
double atof(const char *s);
float atoff(const char *s);
```

Description

`atof` converts the initial portion of a string to a `double`. `atoff` converts the initial portion of a string to a `float`.

The functions parse the character string *s*, locating a substring which can be converted to a floating point value. The substring must match the format:

```
[+|-]digits[.][digits][(e|E)[+|-]digits]
```

The substring converted is the longest initial fragment of *s* that has the expected format, beginning with the first non-whitespace character. The substring is empty if `str` is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit.

`atof(s)` is implemented as `strtod(s, NULL)`. `atoff(s)` is implemented as `strtodf(s, NULL)`.

Returns

`atof` returns the converted substring value, if any, as a `double`; or `0.0`, if no conversion could be performed. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in `errno`. If the correct value would cause underflow, `0.0` is returned and `ERANGE` is stored in `errno`.

`atoff` obeys the same rules as `atof`, except that it returns a `float`.

Portability

`atof` is ANSI C. `atof`, `atoi`, and `atol` are subsumed by `strod` and `strol`, but are used extensively in existing code. These functions are less reliable, but may be faster if the argument is verified to be in a valid range.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.6 `atoi`, `atol`—string to integer

Synopsis

```
#include <stdlib.h>
int atoi(const char *s);
long atol(const char *s);
```

Description

`atoi` converts the initial portion of a string to an `int`. `atol` converts the initial portion of a string to a `long`.

`atoi(s)` is implemented as `(int)strtol(s, NULL, 10).` `atol(s)` is implemented as `strtol(s, NULL, 10).`

Returns

The functions return the converted value, if any. If no conversion was made, `0` is returned.

Portability

`atoi` is ANSI.

No supporting OS subroutines are required.

1.7 `atol`—convert string to long

Synopsis

```
long atol(const char *s);
```

Description

`atol` converts the initial portion of a string to an `long`.

`atol(s)` is implemented as `strtol(s,NULL,10)`.

Portability

`atol` is ANSI.

No supporting OS subroutines are required.

1.8 `bsearch`—binary search

Synopsis

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
    size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

Description

`bsearch` searches an array beginning at *base* for any element that matches *key*, using binary search. *nmemb* is the element count of the array; *size* is the size of each element.

The array must be sorted in ascending order with respect to the comparison function *compar* (which you supply as the last argument of `bsearch`).

You must define the comparison function `(*compar)` to have two arguments; its result must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where “less than” and “greater than” refer to whatever arbitrary ordering is appropriate).

Returns

Returns a pointer to an element of *array* that matches *key*. If more than one matching element is available, the result may point to any of them.

Portability

`bsearch` is ANSI.

No supporting OS subroutines are required.

1.9 `calloc`—allocate space for arrays

Synopsis

```
#include <stdlib.h>
void *calloc(size_t n, size_t s);
void *calloc_r(void *reent, size_t <n>, <size_t> s);
```

Description

Use `calloc` to request a block of memory sufficient to hold an array of *n* elements, each of which has size *s*.

The memory allocated by `calloc` comes out of the same memory pool used by `malloc`, but the memory block is initialized to all zero bytes. (To avoid the overhead of initializing the space, use `malloc` instead.)

The alternate functios `_calloc_r` is reentrant. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, a pointer to the newly allocated space.

If unsuccessful, `NULL`.

Portability

`calloc` is ANSI.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.10 div—divide two integers

Synopsis

```
#include <stdlib.h>
div_t div(int n, int d);
```

Description

Divide n/d, returning quotient and remainder as two integers in a structure `div_t`.

Returns

The result is represented with the structure

```
typedef struct
{
 int quot;
 int rem;
} div_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero *d*, if ‘*r* = div(*n*,*d*);’ then *n* equals ‘*r*.rem + *d* * *r*.quot’.

When *d* is zero, the `quot` member of the result has the same sign as *n* and the largest representable magnitude.

To divide `long` rather than `int` values, use the similar function `ldiv`.

Portability

`div` is ANSI, but the behavior for zero *d* is not specified by the standard.

No supporting OS subroutines are required.

1.11 ecvt,ecvtf,fcvt,fcvtf—double or float to string

Synopsis

```
#include <stdlib.h>

char *ecvt(double val, int chars, int *decpt, int *sgn);
char *ecvtf(float val, int chars, int *decpt, int *sgn);

char *fcvt(double val, int decimals,
    int *decpt, int *sgn);
char *fcvtf(float val, int decimals,
    int *decpt, int *sgn);
```

Description

`ecvt` and `fcvt` produce (null-terminated) strings of digits representating the `double` number *val*. `ecvtf` and `fcvtf` produce the corresponding character representations of `float` numbers.

(The `stdlib` functions `ecvtbuf` and `fcvtbuf` are reentrant versions of `ecvt` and `fcvt`.)

The only difference between `ecvt` and `fcvt` is the interpretation of the second argument (*chars* or *decimals*). For `ecvt`, the second argument *chars* specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvt`, the second argument *decimals* specifies the number of characters to write after the decimal point; all digits for the integer part of *val* are always included.

Since `ecvt` and `fcvt` write only digits in the output string, they record the location of the decimal point in **decpt*, and the sign of the number in **sgn*. After formatting a number, **decpt* contains the number of digits to the left of the decimal point. **sgn* contains `0` if the number is positive, and `1` if it is negative.

Returns

All four functions return a pointer to the new string containing a character representation of *val*.

Portability

None of these functions are ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.12 gvcvt, gcvtf—format double or float as string

Synopsis

```
#include <stdlib.h>

char *gcvt(double val, int precision, char *buf);
char *gcvtf(float val, int precision, char *buf);
```

Description

gcvt writes a fully formatted number as a null-terminated string in the buffer **buf*. gdvtf produces corresponding character representations of float numbers.

gcvt uses the same rules as the printf format '%.*precision*g'—only negative values are signed (with '-'), and either exponential or ordinary decimal-fraction format is chosen depending on the number of significant digits (specified by *precision*).

Returns

The result is a pointer to the formatted representation of *val* (the same as the argument *buf*).

Portability

Neither function is ANSI C.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.

1.13 `ecvtbuf`, `fcvtbuf`—double or float to string

Synopsis

```
#include <stdio.h>

char *ecvtbuf(double val, int chars, int *decpt,
    int *sgn, char *buf);

char *fcvtbuf(double val, int decimals, int *decpt,
    int *sgn, char *buf);
```

Description

`ecvtbuf` and `fcvtbuf` produce (null-terminated) strings of digits representating the `double` number *val*.

The only difference between `ecvtbuf` and `fcvtbuf` is the interpretation of the second argument (*chars* or *decimals*). For `ecvtbuf`, the second argument *chars* specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvtbuf`, the second argument *decimals* specifies the number of characters to write after the decimal point; all digits for the integer part of *val* are always included.

Since `ecvtbuf` and `fcvtbuf` write only digits in the output string, they record the location of the decimal point in `*`*decpt*, and the sign of the number in `*`*sgn*. After formatting a number, `*`*decpt* contains the number of digits to the left of the decimal point. `*`*sgn* contains `0` if the number is positive, and `1` if it is negative. For both functions, you supply a pointer *buf* to an area of memory to hold the converted string.

Returns

Both functions return a pointer to *buf*, the string containing a character representation of *val*.

Portability

Neither function is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.14 exit—end program execution

Synopsis

```
#include &lt;stdlib.h&gt;
void exit(int code);
```

Description

Use `exit` to return control from a program to the host operating environment. Use the argument *code* to pass an exit status to the operating environment: two particular values, `EXIT_SUCCESS` and `EXIT_FAILURE`, are defined in ‘`stdlib.h`’ to indicate success or failure in a portable fashion.

`exit` does two kinds of cleanup before ending execution of your program. First, it calls all application-defined cleanup functions you have enrolled with `atexit`. Second, files and streams are cleaned up: any pending output is delivered to the host system, each open file or stream is closed, and files created by `tmpfile` are deleted.

Returns

`exit` does not return to its caller.

Portability

ANSI C requires `exit`, and specifies that `EXIT_SUCCESS` and `EXIT_FAILURE` must be defined.

Supporting OS subroutines required: `_exit`.

1.15 `getenv`—look up environment variable

Synopsis

```
#include <stdlib.h>
char *getenv(const char *name);
```

Description

`getenv` searches the list of environment variable names and values (using the global pointer '`char **environ`') for a variable whose name matches the string at *`name`*. If a variable name matches, `getenv` returns a pointer to the associated value.

Returns

A pointer to the (string) value of the environment variable, or `NULL` if there is no such environment variable.

Portability

`getenv` is ANSI, but the rules for properly forming names of environment variables vary from one system to another.

`getenv` requires a global pointer `environ`.

1.16 `labs`—long integer absolute value

Synopsis

```
#include <stdlib.h>
long labs(long i);
```

Description

`labs` returns $|x|$, the absolute value of *i* (also called the magnitude of *i*). That is, if *i* is negative, the result is the opposite of *i*, but if *i* is nonnegative the result is *i*.

The similar function `abs` uses and returns `int` rather than `long` values.

Returns

The result is a nonnegative long integer.

Portability

`labs` is ANSI.

No supporting OS subroutine calls are required.

1.17 `ldiv`—divide two long integers

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long n, long d);
```

Description

Divide n/d, returning quotient and remainder as two integers in a structure `ldiv_t`.

Returns

The result is represented with the structure

```
typedef struct
{
 long quot;
 long rem;
} ldiv_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero *d*, if '`r = ldiv(n,d);`' then *n* equals '`r.rem + d*r.quot`'.

When *d* is zero, the `quot` member of the result has the same sign as *n* and the largest representable magnitude.

To divide `int` rather than `long` values, use the similar function `div`.

Portability

`ldiv` is ANSI, but the behavior for zero *d* is not specified by the standard.

No supporting OS subroutines are required.

1.18 `malloc`, `realloc`, `free`—manage memory

Synopsis

```
#include <stdlib.h>
void *malloc(size_t nbytes);
void *realloc(void *aptr, size_t nbytes);
void free(void *aptr);

void *_malloc_r(void *reent, size_t nbytes);
void *_realloc_r(void *reent,
    void *aptr, size_t nbytes);
void _free_r(void *reent, void *aptr);
```

Description

These functions manage a pool of system memory.

Use `malloc` to request allocation of an object with at least *nbytes* bytes of storage available. If the space is available, `malloc` returns a pointer to a newly allocated block as its result.

If you already have a block of storage allocated by `malloc`, but you no longer need all the space allocated to it, you can make it smaller by calling `realloc` with both the object pointer and the new desired size as arguments. `realloc` guarantees that the contents of the smaller object match the beginning of the original object.

Similarly, if you need more space for an object, use `realloc` to request the larger size; again, `realloc` guarantees that the beginning of the new, larger object matches the contents of the original object.

When you no longer need an object originally allocated by `malloc` or `realloc` (or the related function `calloc`), return it to the memory storage pool by calling `free` with the address of the object as the argument. You can also use `realloc` for this purpose by calling it with `0` as the *nbytes* argument.

The alternate functions `_malloc_r`, `_realloc_r`, and `_free_r` are reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`malloc` returns a pointer to the newly allocated space, if successful; otherwise it returns `NULL`. If your application needs to generate empty objects, you may use `malloc(0)` for this purpose.

`realloc` returns a pointer to the new block of memory, or `NULL` if a new block could not be allocated. `NULL` is also the result when you use ‘`realloc(`*aptr*`,0)`’ (which has the same effect as ‘`free(`*aptr*`)`’). You

should always check the result of `realloc`; successful reallocation is not guaranteed even when you request a smaller object.

`free` does not return a result.

Portability

`malloc`, `realloc`, and `free` are specified by the ANSI C standard, but other conforming implementations of `malloc` may behave differently when *nbytes* is zero.

Supporting OS subroutines required: `sbrk`, `write` (if WARN_VLIMIT).

1.19 `mbtowc`—minimal multibyte to wide char converter

Synopsis

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

This is a minimal ANSI-conforming implementation of `mbtowc`. The only "multi-byte character sequences" recognized are single bytes, and they are "converted" to themselves.

Each call to `mbtowc` copies one character from `*s` to `*pwc`, unless *s* is a null pointer.

In this implementation, the argument *n* is ignored.

Returns

This implementation of `mbtowc` returns `0` if *s* is `NULL`; it returns `1` otherwise (reporting the length of the character "sequence" used).

Portability

`mbtowc` is required in the ANSI C standard. However, the precise effects vary with the locale.

`mbtowc` requires no supporting OS subroutines.

1.20 `qsort`—sort an array

Synopsis

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *) );
```

Description

`qsort` sorts an array (beginning at *base*) of *nmemb* objects. *size* describes the size of each element of the array.

You must supply a pointer to a comparison function, using the argument shown as *compar*. (This permits sorting objects of unknown properties.) Define the comparison function to accept two arguments, each a pointer to an element of the array starting at *base*. The result of `(*compar)` must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where "less than" and "greater than" refer to whatever arbitrary ordering is appropriate).

The array is sorted in place; that is, when `qsort` returns, the array elements beginning at *base* have been reordered.

Returns

`qsort` does not return a result.

Portability

`qsort` is required by ANSI (without specifying the sorting algorithm).

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.21 rand, srand—pseudo-random numbers

Synopsis

```
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);

int _rand_r(void *reent);
void _srand_r(void *reent, unsigned int seed);
```

Description

rand returns a different integer each time it is called; each integer is chosen by an algorithm designed to be unpredictable, so that you can use rand when you require a random number. The algorithm depends on a static variable called the "random seed"; starting with a given value of the random seed always produces the same sequence of numbers in successive calls to rand.

You can set the random seed using srand; it does nothing beyond storing its argument in the static variable used by rand. You can exploit this to make the pseudo-random sequence less predictable, if you wish, by using some other unpredictable value (often the least significant parts of a time-varying value) as the random seed before beginning a sequence of calls to rand; or, if you wish to ensure (for example, while debugging) that successive runs of your program use the same "random" numbers, you can use srand to set the same random seed at the outset.

_rand_r and _srand_r are reentrant versions of rand and srand. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

rand returns the next pseudo-random integer in sequence; it is a number between 0 and RAND_MAX (inclusive).

srand does not return a result.

Portability

rand is required by ANSI, but the algorithm for pseudo-random number generation is not specified; therefore, even if you use the same random seed, you cannot expect the same sequence of results on two different systems.

rand requires no supporting OS subroutines.

1.22 `strtod`, `strtodf`—string to double or float

Synopsis

```
#include <stdlib.h>
double strtod(const char *str, char **tail);
float strtodf(const char *str, char **tail);

double _strtod_r(void *reent,
    const char *str, char **tail);
```

Description

The function `strtod` parses the character string *str*, producing a substring which can be converted to a double value. The substring converted is the longest initial subsequence of *str*, beginning with the first non-whitespace character, that has the format:

`[+|-]`*digits*`[.][`*digits*`][(e|E)[+|-]`*digits*`]`

The substring contains no characters if *str* is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit. If the substring is empty, no conversion is done, and the value of *str* is stored in `*`*tail*. Otherwise, the substring is converted, and a pointer to the final string (which will contain at least the terminating null character of *str*) is stored in `*`*tail*. If you want no assignment to `*`*tail*, pass a null pointer as *tail*. `strtodf` is identical to `strtod` except for its return type.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule.

The alternate function `_strtod_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`strtod` returns the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in errno. If the correct value would cause underflow, 0 is returned and `ERANGE` is stored in errno.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

1.23 strtol—string to long

Synopsis

```
#include <stdlib.h>
long strtol(const char *s, char **ptr,int base);

long _strtol_r(void *reent,
    const char *s, char **ptr,int base);
```

Description

The function strtol converts the string *s to a long. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by *base*; and a trailing portion consisting of zero or more unparseable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a long and returns the result.

If the value of *base* is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘0x’ indicating a hexadecimal base, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *base*, with an optional plus or minus sign. The letters a–z (or, equivalently, A–Z) are used to signify values from 10 to 35; only letters whose ascribed values are less than *base* are permitted. If *base* is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, strtol attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not NULL.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not NULL).

The alternate function _strtol_r is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`strtol` returns the converted value, if any. If no conversion was made, 0 is returned.

`strtol` returns `LONG_MAX` or `LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability

`strtol` is ANSI.

No supporting OS subroutines are required.

1.24 strtoul—string to unsigned long

Synopsis

```
#include <stdlib.h>
unsigned long strtoul(const char *s, char **ptr,
    int base);

unsigned long _strtoul_r(void *reent, const char *s,
    char **ptr, int base);
```

Description

The function strtoul converts the string *s to an unsigned long. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by *base* (for example, 0 through 7 if the value of *base* is 8); and a trailing portion consisting of one or more unparseable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long integer, and returns the result.

If the value of *base* is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible 0x indicating hexadecimal radix, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by *base*. The letters a–z (or A–Z) are used as digits valued from 10 to 35. If *base* is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, strtoul attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not NULL.

If the subject string is empty (that is, if *s does not start with a substring in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not NULL).

The alternate function _strtoul_r is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`strtoul` returns the converted value, if any. If no conversion was made, `0` is returned.

`strtoul` returns `ULONG_MAX` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability

`strtoul` is ANSI.

`strtoul` requires no supporting OS subroutines.

1.25 `system`—execute command string

Synopsis

```
#include <stdlib.h>
int system(char *s);

int _system_r(void *reent, char *s);
```

Description

Use `system` to pass a command string `*s` to `/bin/sh` on your system, and wait for it to finish executing.

Use ‘`system(NULL)`’ to test whether your system has `/bin/sh` available.

The alternate function `_system_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`system(NULL)` returns a non-zero value if `/bin/sh` is available, and `0` if it is not.

With a command argument, the result of `system` is the exit status returned by `/bin/sh`.

Portability

ANSI C requires `system`, but leaves the nature and effects of a command processor undefined. ANSI C does, however, specify that `system(NULL)` return zero or nonzero to report on the existence of a command processor.

POSIX.2 requires `system`, and requires that it invoke `/bin/sh`.

Supporting OS subroutines required: `_exit`, `execve`, `fork`, `wait`.

1.26 `wctomb`—minimal wide char to multibyte converter

Synopsis

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

This is a minimal ANSI-conforming implementation of `wctomb`. The only "wide characters" recognized are single bytes, and they are "converted" to themselves.

Each call to `wctomb` copies the character *wchar* to `*`*s*, unless *s* is a null pointer.

Returns

This implementation of `wctomb` returns `0` if *s* is `NULL`; it returns `1` otherwise (reporting the length of the character "sequence" generated).

Portability

`wctomb` is required in the ANSI C standard. However, the precise effects vary with the locale.

`wctomb` requires no supporting OS subroutines.

2 Character Type Macros and Functions (‘`ctype.h`’)

This chapter groups macros (which are also available as subroutines) to classify characters into several categories (alphabetic, numeric, control characters, whitespace, and so on), or to perform simple character mappings.

The header file ‘`ctype.h`’ defines the macros.

2.1 `isalnum`—alphanumeric character predicate

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Description

`isalnum` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for alphabetic or numeric ASCII characters, and `0` for other arguments. It is defined for all integer values.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isalnum`'.

Returns

`isalnum` returns non-zero if *c* is a letter (`a`–`z` or `A`–`Z`) or a digit (`0`–`9`).

Portability

`isalnum` is ANSI C.

No OS subroutines are required.

2.2 isalpha—alphabetic character predicate

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Description

isalpha is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero when *c* represents an alphabetic ASCII character, and 0 otherwise. It is defined only when isascii(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘#undef isalpha’.

Returns

isalpha returns non-zero if *c* is a letter (A–Z or a–z).

Portability

isalpha is ANSI C.

No supporting OS subroutines are required.

2.3 `isascii`—ASCII character predicate

Synopsis

```
#include <ctype.h>
int isascii(int c);
```

Description

`isascii` is a macro which returns non-zero when *c* is an ASCII character, and 0 otherwise. It is defined for all integer values.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isascii`'.

Returns

`isascii` returns non-zero if the low order byte of *c* is in the range 0 to 127 (`0x00`–`0x7F`).

Portability

`isascii` is ANSI C.

No supporting OS subroutines are required.

2.4 iscntrl—control character predicate

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

Description

`iscntrl` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for control characters, and 0 for other characters. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘`#undef iscntrl`’.

Returns

`iscntrl` returns non-zero if *c* is a delete character or ordinary control character (`0x7F` or `0x00`–`0x1F`).

Portability

`iscntrl` is ANSI C.

No supporting OS subroutines are required.

2.5 `isdigit`—decimal digit predicate

Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Description

`isdigit` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for decimal digits, and 0 for other characters. It is defined only when `isascii(c)` is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isdigit`'.

Returns

`isdigit` returns non-zero if *c* is a decimal digit (`0`–`9`).

Portability

`isdigit` is ANSI C.

No supporting OS subroutines are required.

2.6 `islower`—lower-case character predicate

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

`islower` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for minuscules (lower-case alphabetic characters), and 0 for other characters. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘`#undef islower`’.

Returns

`islower` returns non-zero if *c* is a lower case letter (`a`–`z`).

Portability

`islower` is ANSI C.

No supporting OS subroutines are required.

2.7 `isprint`, `isgraph`—printable character predicates

Synopsis

```
#include <ctype.h>
int isprint(int c);
int isgraph(int c);
```

Description

`isprint` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable characters, and 0 for other character arguments. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining either macro using '`#undef isprint`' or '`#undef isgraph`'.

Returns

`isprint` returns non-zero if *c* is a printing character, (`0x20`–`0x7E`). `isgraph` behaves identically to `isprint`, except that the space character (`0x20`) is excluded.

Portability

`isprint` and `isgraph` are ANSI C.

No supporting OS subroutines are required.

2.8 `ispunct`—punctuation character predicate

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

`ispunct` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable punctuation characters, and 0 for other characters. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef ispunct`'.

Returns

`ispunct` returns non-zero if *c* is a printable punctuation character (`isgraph(c) && !isalnum(c)`).

Portability

`ispunct` is ANSI C.

No supporting OS subroutines are required.

2.9 `isspace`—whitespace character predicate

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Description

`isspace` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for whitespace characters, and 0 for other characters. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isspace`'.

Returns

`isspace` returns non-zero if *c* is a space, tab, carriage return, new line, vertical tab, or formfeed (`0x09`–`0x0D`, `0x20`).

Portability

`isspace` is ANSI C.

No supporting OS subroutines are required.

2.10 `isupper`—uppercase character predicate

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Description

`isupper` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for upper-case letters (`A–Z`), and 0 for other characters. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘`#undef isupper`’.

Returns

`isupper` returns non-zero if *c* is a upper case letter (A-Z).

Portability

`isupper` is ANSI C.

No supporting OS subroutines are required.

2.11 `isxdigit`—hexadecimal digit predicate

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Description

`isxdigit` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for hexadecimal digits, and `0` for other characters. It is defined only when `isascii`(*c*) is true or *c* is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isxdigit`'.

Returns

`isxdigit` returns non-zero if *c* is a hexadecimal digit (`0`–`9`, `a`–`f`, or `A`–`F`).

Portability

`isxdigit` is ANSI C.

No supporting OS subroutines are required.

2.12 `toascii`—force integers to ASCII range

Synopsis

```
#include <ctype.h>
int toascii(int c);
```

Description

`toascii` is a macro which coerces integers to the ASCII range (0–127) by zeroing any higher-order bits.

You can use a compiled subroutine instead of the macro definition by undefining this macro using ‘`#undef toascii`’.

Returns

`toascii` returns integers between 0 and 127.

Portability

`toascii` is not ANSI C.

No supporting OS subroutines are required.

2.13 `tolower`—translate characters to lower case

Synopsis

```
#include <ctype.h>
int tolower(int c);
int _tolower(int c);
```

Description

`tolower` is a macro which converts upper-case characters to lower case, leaving all other characters unchanged. It is only defined when *c* is an integer in the range `EOF` to `255`.

You can use a compiled subroutine instead of the macro definition by undefining this macro using '`#undef tolower`'.

`_tolower` performs the same conversion as `tolower`, but should only be used when *c* is known to be an uppercase character (`A`–`Z`).

Returns

`tolower` returns the lower-case equivalent of *c* when it is a character between `A` and `Z`, and *c* otherwise.

`_tolower` returns the lower-case equivalent of *c* when it is a character between `A` and `Z`. If *c* is not one of these characters, the behaviour of `_tolower` is undefined.

Portability

`tolower` is ANSI C. `_tolower` is not recommended for portable programs.

No supporting OS subroutines are required.

2.14 toupper—translate characters to upper case

Synopsis

```
#include <ctype.h>
int toupper(int c);
int _toupper(int c);
```

Description

toupper is a macro which converts lower-case characters to upper case, leaving all other characters unchanged. It is only defined when *c* is an integer in the range EOF to 255.

You can use a compiled subroutine instead of the macro definition by undefining this macro using ‘#undef toupper’.

_toupper performs the same conversion as toupper, but should only be used when *c* is known to be a lowercase character (a–z).

Returns

toupper returns the upper-case equivalent of *c* when it is a character between a and z, and *c* otherwise.

_toupper returns the upper-case equivalent of *c* when it is a character between a and z. If *c* is not one of these characters, the behaviour of _toupper is undefined.

Portability

toupper is ANSI C. _toupper is not recommended for portable programs.

No supporting OS subroutines are required.

3 Input and Output (‘stdio.h’)

This chapter comprises functions to manage files or other input/output streams. Among these functions are subroutines to generate or scan strings according to specifications from a format string.

The underlying facilities for input and output depend on the host system, but these functions provide a uniform interface.

The corresponding declarations are in ‘stdio.h’.

The reentrant versions of these functions use macros

```
_stdin_r(reent)
_stdout_r(reent)
_stderr_r(reent)
```

instead of the globals stdin, stdout, and stderr. The argument <[reent]> is a pointer to a reentrancy structure.

3.1 `clearerr`—clear file or stream error indicator

Synopsis

```
#include <stdio.h>
void clearerr(FILE *fp);
```

Description

The `stdio` functions maintain an error indicator with each file pointer *fp*, to record whether any read or write errors have occurred on the associated file or stream. Similarly, it maintains an end-of-file indicator to record whether there is no more data in the file.

Use `clearerr` to reset both of these indicators.

See `ferror` and `feof` to query the two indicators.

Returns

`clearerr` does not return a result.

Portability

ANSI C requires `clearerr`.

No supporting OS subroutines are required.

3.2 `fclose`—close a file

Synopsis

```
#include <stdio.h>
int fclose(FILE *fp);
```

Description

If the file or stream identified by *fp* is open, `fclose` closes it, after first ensuring that any pending data is written (by calling `fflush(fp)`).

Returns

`fclose` returns `0` if successful (including when *fp* is `NULL` or not an open file); otherwise, it returns `EOF`.

Portability

`fclose` is required by ANSI C.

Required OS subroutines: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.3 `feof`—test for end of file

Synopsis

```
#include <stdio.h>
int feof(FILE *fp);
```

Description

`feof` tests whether or not the end of the file identified by *fp* has been reached.

Returns

`feof` returns `0` if the end of file has not yet been reached; if at end of file, the result is nonzero.

Portability

`feof` is required by ANSI C.

No supporting OS subroutines are required.

3.4 `ferror`—test whether read/write error has occurred

Synopsis

```
#include <stdio.h>
int ferror(FILE *fp);
```

Description

The `stdio` functions maintain an error indicator with each file pointer *fp*, to record whether any read or write errors have occurred on the associated file or stream. Use `ferror` to query this indicator.

See `clearerr` to reset the error indicator.

Returns

`ferror` returns `0` if no errors have occurred; it returns a nonzero value otherwise.

Portability

ANSI C requires `ferror`.

No supporting OS subroutines are required.

3.5 `fflush`—flush buffered file output

Synopsis

```
#include <stdio.h>
int fflush(FILE *fp);
```

Description

The `stdio` output functions can buffer output before delivering it to the host system, in order to minimize the overhead of system calls.

Use `fflush` to deliver any such pending output (for the file or stream identified by *fp*) to the host system.

If *fp* is `NULL`, `fflush` delivers pending output from all open files.

Returns

`fflush` returns `0` unless it encounters a write error; in that situation, it returns `EOF`.

Portability

ANSI C requires `fflush`.

No supporting OS subroutines are required.

3.6 fgetc—get a character from a file or stream

Synopsis

```
#include <stdio.h>
int fgetc(FILE *fp);
```

Description

Use `fgetc` to get the next single character from the file or stream identified by *fp*. As a side effect, `fgetc` advances the file's current position indicator.

For a macro version of this function, see `getc`.

Returns

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `fgetc` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using the `ferror` and `feof` functions.

Portability

ANSI C requires `fgetc`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.7 `fgetpos`—record position in a stream or file

Synopsis

```
#include <stdio.h>
int fgetpos(FILE *fp, fpos_t *pos);
```

Description

Objects of type `FILE` can have a "position" that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

You can use `fgetpos` to report on the current position for a file identified by *fp*; `fgetpos` will write a value representing that position at `*`*pos*. Later, you can use this value with `fsetpos` to return the file to this position.

In the current implementation, `fgetpos` simply uses a character count to represent the file position; this is the same number that would be returned by `ftell`.

Returns

`fgetpos` returns `0` when successful. If `fgetpos` fails, the result is `1`. Failure occurs on streams that do not support positioning; the global `errno` indicates this condition with the value `ESPIPE`.

Portability

`fgetpos` is required by the ANSI C standard, but the meaning of the value it records is not specified beyond requiring that it be acceptable as an argument to `fsetpos`. In particular, other conforming C implementations may return a different result from `ftell` than what `fgetpos` writes at `*`*pos*.

No supporting OS subroutines are required.

3.8 `fgets`—get character string from a file or stream

Synopsis

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
```

Description

Reads at most *n-1* characters from *fp* until a newline is found. The characters including to the newline are stored in *buf*. The buffer is terminated with a 0.

Returns

`fgets` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If no data are read, NULL is returned instead.

Portability

`fgets` should replace all uses of `gets`. Note however that `fgets` returns all of the data, while `gets` removes the trailing newline (with no indication that it has done so.)

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.9 `fiprintf`—format output to file (integer only)

Synopsis

```
#include <stdio.h>

int fiprintf(FILE *fd, const char *format, ...);
```

Description

`fiprintf` is a restricted version of `fprintf`: it has the same arguments and behavior, save that it cannot perform any floating-point formatting—the `f`, `g`, `G`, `e`, and `F` type specifiers are not recognized.

Returns

`fiprintf` returns the number of bytes in the output string, save that the concluding `NULL` is not counted. `fiprintf` returns when the end of the format string is encountered. If an error occurs, `fiprintf` returns `EOF`.

Portability

`fiprintf` is not required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.10 fopen—open a file

Synopsis

```
#include <stdio.h>
FILE *fopen(const char *file, const char *mode);

FILE *_fopen_r(void *reent,
    const char *file, const char *mode);
```

Description

fopen initializes the data structures needed to read or write a file. Specify the file’s name as the string at *file*, and the kind of access you need to the file with the string at *mode*.

The alternate function _fopen_r is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Three fundamental kinds of access are available: read, write, and append. **mode* must begin with one of the three characters ‘r’, ‘w’, or ‘a’, to select one of these:

r
: Open the file for reading; the operation will fail if the file does not exist, or if the host system does not permit you to read it.

w
: Open the file for writing *from the beginning* of the file: effectively, this always creates a new file. If the file whose name you specified already existed, its old contents are discarded.

a
: Open the file for appending data, that is writing from the end of file. When you open a file this way, all data always goes to the current end of file; you cannot change this using fseek.

Some host systems distinguish between “binary” and “text” files. Such systems may perform data transformations on data written to, or read from, files opened as “text”. If your system is one of these, then you can append a ‘b’ to any of the three modes above, to specify that you are opening the file as a binary file (the default is to open the file as a text file).

‘rb’, then, means “read binary”; ‘wb’, “write binary”; and ‘ab’, “append binary”.

To make C programs more portable, the ‘b’ is accepted on all systems, whether or not it makes a difference.

Finally, you might need to both read and write from the same file. You can also append a ‘+’ to any of the three modes, to permit this. (If you want to append both ‘b’ and ‘+’, you can do it in either order: for example, "rb+" means the same thing as "r+b" when used as a mode string.)

Use `"r+"` (or `"rb+"`) to permit reading and writing anywhere in an existing file, without discarding any data; `"w+"` (or `"wb+"`) to create a new file (or begin by discarding all data from an old one) that permits reading and writing anywhere in it; and `"a+"` (or `"ab+"`) to permit reading anywhere in an existing file, but writing only at the end.

Returns

`fopen` returns a file pointer which you can use for other file operations, unless the file you requested could not be opened; in that situation, the result is `NULL`. If the reason for failure was an invalid string at *mode*, `errno` is set to `EINVAL`.

Portability

`fopen` is required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

3.11 `fdopen`—turn open file into a stream

Synopsis

```
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
FILE *_fdopen_r(void *reent,
    int fd, const char *mode);
```

Description

`fdopen` produces a file descriptor of type `FILE *`, from a descriptor for an already-open file (returned, for example, by the system subroutine `open` rather than by `fopen`). The *mode* argument has the same meanings as in `fopen`.

Returns

File pointer or `NULL`, as for `fopen`.

Portability

`fdopen` is ANSI.

3.12 `fputc`—write a character on a stream or file

Synopsis

```
#include <stdio.h>
int fputc(int ch, FILE *fp);
```

Description

`fputc` converts the argument *ch* from an `int` to an `unsigned char`, then writes it to the file or stream identified by *fp*.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator oadvances by one.

For a macro version of this function, see `putc`.

Returns

If successful, `fputc` returns its argument *ch*. If an error intervenes, the result is `EOF`. You can use '`ferror(fp)`' to query for errors.

Portability

`fputc` is required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.13 `fputs`—write a character string in a file or stream

Synopsis

```
#include <stdio.h>
int fputs(const char *s, FILE *fp);
```

Description

`fputs` writes the string at *s* (but without the trailing null) to the file or stream identified by *fp*.

Returns

If successful, the result is `0`; otherwise, the result is `EOF`.

Portability

ANSI C requires `fputs`, but does not specify that the result on success must be `0`; any non-negative value is permitted.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.14 `fread`—read array elements from a file

Synopsis

```
#include <stdio.h>
size_t fread(void *buf, size_t size, size_t count,
    FILE *fp);
```

Description

`fread` attempts to copy, from the file or stream identified by *fp*, *count* elements (each of size *size*) into memory, starting at *buf*. `fread` may copy fewer elements than *count* if an error, or end of file, intervenes.

`fread` also advances the file position indicator (if any) for *fp* by the number of *characters* actually read.

Returns

The result of `fread` is the number of elements it succeeded in reading.

Portability

ANSI C requires `fread`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.15 `freopen`—open a file using an existing file descriptor

Synopsis

```
#include <stdio.h>
FILE *freopen(const char *file, const char *mode,
    FILE *fp);
```

Description

Use this variant of `fopen` if you wish to specify a particular file descriptor *fp* (notably `stdin`, `stdout`, or `stderr`) for the file.

If *fp* was associated with another file or stream, `freopen` closes that other file or stream (but ignores any errors while closing it).

file and *mode* are used just as in `fopen`.

Returns

If successful, the result is the same as the argument *fp*. If the file cannot be opened as specified, the result is `NULL`.

Portability

ANSI C requires `freopen`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

3.16 `fseek`—set file position

Synopsis

```
#include <stdio.h>
int fseek(FILE *fp, long offset, int whence)
```

Description

Objects of type `FILE` can have a “position” that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

You can use `fseek` to set the position for the file identified by *fp*. The value of *offset* determines the new position, in one of three ways selected by the value of *whence* (defined as macros in ‘`stdio.h`’):

`SEEK_SET`—*offset* is the absolute file position (an offset from the beginning of the file) desired. *offset* must be positive.

`SEEK_CUR`—*offset* is relative to the current file position. *offset* can meaningfully be either positive or negative.

`SEEK_END`—*offset* is relative to the current end of file. *offset* can meaningfully be either positive (to increase the size of the file) or negative.

See `ftell` to determine the current file position.

Returns

`fseek` returns `0` when successful. If `fseek` fails, the result is `EOF`. The reason for failure is indicated in `errno`: either `ESPIPE` (the stream identified by *fp* doesn’t support repositioning) or `EINVAL` (invalid file position).

Portability

ANSI C requires `fseek`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.17 fsetpos—restore position of a stream or file

Synopsis

```
#include <stdio.h>
int fsetpos(FILE *fp, const fpos_t *pos);
```

Description

Objects of type `FILE` can have a "position" that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

You can use `fsetpos` to return the file identified by *fp* to a previous position `*`*pos* (after first recording it with `fgetpos`).

See `fseek` for a similar facility.

Returns

`fgetpos` returns `0` when successful. If `fgetpos` fails, the result is `1`. The reason for failure is indicated in `errno`: either `ESPIPE` (the stream identified by *fp* doesn't support repositioning) or `EINVAL` (invalid file position).

Portability

ANSI C requires `fsetpos`, but does not specify the nature of `*`*pos* beyond identifying it as written by `fgetpos`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.18 `ftell`—return position in a stream or file

Synopsis

```
#include <stdio.h>
long ftell(FILE *fp);
```

Description

Objects of type `FILE` can have a “position” that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

The result of `ftell` is the current position for a file identified by *fp*. If you record this result, you can later use it with `fseek` to return the file to this position.

In the current implementation, `ftell` simply uses a character count to represent the file position; this is the same number that would be recorded by `fgetpos`.

Returns

`ftell` returns the file position, if possible. If it cannot do this, it returns `-1L`. Failure occurs on streams that do not support positioning; the global `errno` indicates this condition with the value `ESPIPE`.

Portability

`ftell` is required by the ANSI C standard, but the meaning of its result (when successful) is not specified beyond requiring that it be acceptable as an argument to `fseek`. In particular, other conforming C implementations may return a different result from `ftell` than what `fgetpos` records.

No supporting OS subroutines are required.

3.19 `fwrite`—write array elements

Synopsis

```
#include <stdio.h>
size_t fwrite(const void *buf, size_t size,
    size_t count, FILE *fp);
```

Description

`fwrite` attempts to copy, starting from the memory location *buf*, *count* elements (each of size *size*) into the file or stream identified by *fp*. `fwrite` may copy fewer elements than *count* if an error intervenes.

`fwrite` also advances the file position indicator (if any) for *fp* by the number of *characters* actually written.

Returns

If `fwrite` succeeds in writing all the elements you specify, the result is the same as the argument *count*. In any event, the result is the number of complete elements that `fwrite` copied to the file.

Portability

ANSI C requires `fwrite`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.20 `getc`—read a character (macro)

Synopsis

```
#include <stdio.h>
int getc(FILE *fp);
```

Description

`getc` is a macro, defined in `stdio.h`. You can use `getc` to get the next single character from the file or stream identified by *fp*. As a side effect, `getc` advances the file's current position indicator.

For a subroutine version of this macro, see `fgetc`.

Returns

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `getc` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using the `ferror` and `feof` functions.

Portability

ANSI C requires `getc`; it suggests, but does not require, that `getc` be implemented as a macro. The standard explicitly permits macro implementations of `getc` to use the argument more than once; therefore, in a portable program, you should not use an expression with side effects as the `getc` argument.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.21 getchar—read a character (macro)

Synopsis

```
#include <stdio.h>
int getchar(void);

int _getchar_r(void *reent);
```

Description

getchar is a macro, defined in stdio.h. You can use getchar to get the next single character from the standard input stream. As a side effect, getchar advances the standard input's current position indicator.

The alternate function _getchar_r is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The next character (read as an unsigned char, and cast to int), unless there is no more data, or the host system reports a read error; in either of these situations, getchar returns EOF.

You can distinguish the two situations that cause an EOF result by using ‘ferror(stdin)’ and ‘feof(stdin)’.

Portability

ANSI C requires getchar; it suggests, but does not require, that getchar be implemented as a macro.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.

3.22 `gets`—get character string (obsolete, use `fgets` instead)

Synopsis

```
#include <stdio.h>

char *gets(char *buf);

char *_gets_r(void *reent, char *buf);
```

Description

Reads characters from standard input until a newline is found. The characters up to the newline are stored in *buf*. The newline is discarded, and the buffer is terminated with a 0.

This is a *dangerous* function, as it has no way of checking the amount of space available in *buf*. One of the attacks used by the Internet Worm of 1988 used this to overrun a buffer allocated on the stack of the finger daemon and overwrite the return address, causing the daemon to execute code downloaded into it over the connection.

The alternate function `_gets_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`gets` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If end of file occurs with no data in the buffer, NULL is returned.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.23 `iprintf`—write formatted output (integer only)

Synopsis

```
#include <stdio.h>

int iprintf(const char *format, ...);
```

Description

`iprintf` is a restricted version of `printf`: it has the same arguments and behavior, save that it cannot perform any floating-point formatting: the `f`, `g`, `G`, `e`, and `F` type specifiers are not recognized.

Returns

`iprintf` returns the number of bytes in the output string, save that the concluding `NULL` is not counted. `iprintf` returns when the end of the format string is encountered. If an error occurs, `iprintf` returns `EOF`.

Portability

`iprintf` is not required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.24 `mktemp`, `mkstemp`—generate unused file name

Synopsis

```
#include <stdio.h>
char *mktemp(char *path);
int mkstemp(char *path);

char *_mktemp_r(void *reent, char *path);
int *_mkstemp_r(void *reent, char *path);
```

Description

`mktemp` and `mkstemp` attempt to generate a file name that is not yet in use for any existing file. `mkstemp` creates the file and opens it for reading and writing; `mktemp` simply generates the file name.

You supply a simple pattern for the generated file name, as the string at *path*. The pattern should be a valid filename (including path information if you wish) ending with some number of '`X`' characters. The generated filename will match the leading part of the name you supply, with the trailing '`X`' characters replaced by some combination of digits and letters.

The alternate functions `_mktemp_r` and `_mkstemp_r` are reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`mktemp` returns the pointer *path* to the modified string representing an unused filename, unless it could not generate one, or the pattern you provided is not suitable for a filename; in that case, it returns `NULL`.

`mkstemp` returns a file descriptor to the newly created file, unless it could not generate an unused filename, or the pattern you provided is not suitable for a filename; in that case, it returns `-1`.

Portability

ANSI C does not require either `mktemp` or `mkstemp`; the System V Interface Definition requires `mktemp` as of Issue 2.

Supporting OS subroutines required: `getpid`, `open`, `stat`.

3.25 perror—print an error message on standard error

Synopsis

```
#include <stdio.h>
void perror(char *prefix);

void _perror_r(void *reent, char *prefix);
```

Description

Use `perror` to print (on standard error) an error message corresponding to the current value of the global variable `errno`. Unless you use `NULL` as the value of the argument *prefix*, the error message will begin with the string at *prefix*, followed by a colon and a space (`: `). The remainder of the error message is one of the strings described for `strerror`.

The alternate function `_perror_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`perror` returns no result.

Portability

ANSI C requires `perror`, but the strings issued vary from one implementation to another.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.26 `putc`—write a character (macro)

Synopsis

```
#include <stdio.h>
int putc(int ch, FILE *fp);
```

Description

`putc` is a macro, defined in `stdio.h`. `putc` writes the argument *ch* to the file or stream identified by *fp*, after converting it from an `int` to an `unsigned char`.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator advances by one.

For a subroutine version of this macro, see `fputc`.

Returns

If successful, `putc` returns its argument *ch*. If an error intervenes, the result is `EOF`. You can use '`ferror(fp)`' to query for errors.

Portability

ANSI C requires `putc`; it suggests, but does not require, that `putc` be implemented as a macro. The standard explicitly permits macro implementations of `putc` to use the *fp* argument more than once; therefore, in a portable program, you should not use an expression with side effects as this argument.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.27 `putchar`—write a character (macro)

Synopsis

```
#include <stdio.h>
int putchar(int ch);

int _putchar_r(void *reent, int ch);
```

Description

`putchar` is a macro, defined in `stdio.h`. `putchar` writes its argument to the standard output stream, after converting it from an `int` to an `unsigned char`.

The alternate function `_putchar_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, `putchar` returns its argument *ch*. If an error intervenes, the result is `EOF`. You can use '`ferror(stdin)`' to query for errors.

Portability

ANSI C requires `putchar`; it suggests, but does not require, that `putchar` be implemented as a macro.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.28 `puts`—write a character string

Synopsis

```
#include <stdio.h>
int puts(const char *s);

int _puts_r(void *reent, const char *s);
```

Description

`puts` writes the string at *s* (followed by a newline, instead of the trailing null) to the standard output stream.

The alternate function `_puts_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, the result is `0`; otherwise, the result is `EOF`.

Portability

ANSI C requires `puts`, but does not specify that the result on success must be `0`; any non-negative value is permitted.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.29 remove—delete a file's name

Synopsis

```
#include <stdio.h>
int remove(char *filename);

int _remove_r(void *reent, char *filename);
```

Description

Use `remove` to dissolve the association between a particular filename (the string at *filename*) and the file it represents. After calling `remove` with a particular filename, you will no longer be able to open the file by that name.

In this implementation, you may use `remove` on an open file without error; existing file descriptors for the file will continue to access the file's data until the program using them closes the file.

The alternate function `_remove_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`remove` returns `0` if it succeeds, `-1` if it fails.

Portability

ANSI C requires `remove`, but only specifies that the result on failure be nonzero. The behavior of `remove` when you call it on an open file may vary among implementations.

Supporting OS subroutine required: `unlink`.

3.30 `rename`—rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *old, const char *new);

int _rename_r(void *reent,
    const char *old, const char *new);
```

Description

Use `rename` to establish a new name (the string at *new*) for a file now known by the string at *old*. After a successful `rename`, the file is no longer accessible by the string at *old*.

If `rename` fails, the file named `*old` is unaffected. The conditions for failure depend on the host operating system.

The alternate function `_rename_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The result is either `0` (when successful) or `-1` (when the file could not be renamed).

Portability

ANSI C requires `rename`, but only specifies that the result on failure be nonzero. The effects of using the name of an existing file as `*new` may vary from one implementation to another.

Supporting OS subroutines required: `link`, `unlink`.

3.31 `rewind`—reinitialize a file or stream

Synopsis

```
#include <stdio.h>
void rewind(FILE *fp);
```

Description

`rewind` returns the file position indicator (if any) for the file or stream identified by *fp* to the beginning of the file. It also clears any error indicator and flushes any pending output.

Returns

`rewind` does not return a result.

Portability

ANSI C requires `rewind`.

No supporting OS subroutines are required.

3.32 `setbuf`—specify full buffering for a file or stream

Synopsis

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
```

Description

`setbuf` specifies that output to the file or stream identified by *fp* should be fully buffered. All output for this file will go to a buffer (of size `BUFSIZ`, specified in '`stdio.h`'). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument *buf*. It must have size `BUFSIZ`. You can also use `NULL` as the value of *buf*, to signal that the `setbuf` function is to allocate the buffer.

Warnings

You may only use `setbuf` before performing any file operation other than opening the file.

If you supply a non-null *buf*, you must ensure that the associated storage continues to be available until you close the stream identified by *fp*.

Returns

`setbuf` does not return a result.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `setbuf`. However, they differ on the meaning of a `NULL` buffer pointer: the SVID issue 2 specification says that a `NULL` buffer pointer requests unbuffered output. For maximum portability, avoid `NULL` buffer pointers.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.33 setvbuf—specify file or stream buffering

Synopsis

```
#include <stdio.h>
int setvbuf(FILE *fp, char *buf,
    int mode, size_t size);
```

Description

Use setvbuf to specify what kind of buffering you want for the file or stream identified by *fp*, by using one of the following values (from stdio.h) as the *mode* argument:

_IONBF Do not use a buffer: send output directly to the host system for the file or stream identified by *fp*.

_IOFBF Use full output buffering: output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

_IOLBF Use line buffering: pass on output to the host system at every newline, as well as when the buffer is full, or when an input operation intervenes.

Use the *size* argument to specify how large a buffer you wish. You can supply the buffer itself, if you wish, by passing a pointer to a suitable area of memory as *buf*. Otherwise, you may pass NULL as the *buf* argument, and setvbuf will allocate the buffer.

Warnings

You may only use setvbuf before performing any file operation other than opening the file.

If you supply a non-null *buf*, you must ensure that the associated storage continues to be available until you close the stream identified by *fp*.

Returns

A 0 result indicates success, EOF failure (invalid *mode* or *size* can cause failure).

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require setvbuf. However, they differ on the meaning of a NULL buffer pointer: the SVID issue 2 specification says that a NULL buffer pointer requests unbuffered output. For maximum portability, avoid NULL buffer pointers.

Both specifications describe the result on failure only as a nonzero value.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.34 `siprintf`—write formatted output (integer only)

Synopsis

```
#include <stdio.h>

int siprintf(char *str, const char *format [, arg, ...]);
```

Description

`siprintf` is a restricted version of `sprintf`: it has the same arguments and behavior, save that it cannot perform any floating-point formatting: the `f`, `g`, `G`, `e`, and `F` type specifiers are not recognized.

Returns

`siprintf` returns the number of bytes in the output string, save that the concluding `NULL` is not counted. `siprintf` returns when the end of the format string is encountered.

Portability

`siprintf` is not required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.35 `printf`, `fprintf`, `sprintf`—format output

Synopsis

```
#include <stdio.h>

int printf(const char *format [, arg, ...]);
int fprintf(FILE *fd, const char *format [, arg, ...]);
int sprintf(char *str, const char *format [, arg, ...]);
```

Description

`printf` accepts a series of arguments, applies to each a format specifier from `*format`, and writes the formatted data to `stdout`, terminated with a null character. The behavior of `printf` is undefined if there are not enough arguments for the format. `printf` returns when it reaches the end of the format string. If there are more arguments than the format requires, excess arguments are ignored.

`fprintf` and `sprintf` are identical to `printf`, other than the destination of the formatted output: `fprintf` sends the output to a specified file *fd*, while `sprintf` stores the output in the specified char array *str*. For `sprintf`, the behavior is also undefined if the output `*str` overlaps with one of the arguments. *format* is a pointer to a charater string containing two types of objects: ordinary characters (other than `%`), which are copied unchanged to the output, and conversion specifications, each of which is introduced by `%`. (To include `%` in the output, use `%%` in the format string.) A conversion specification has the following form:

`%[flags][width][.prec][size][type]`

The fields of the conversion specification have the following meanings:

- *flags*

  an optional sequence of characters which control output justification, numeric signs, decimal points, trailing zeroes, and octal and hex prefixes. The flag characters are minus (`-`), plus (`+`), space ( ), zero (`0`), and sharp (`#`). They can appear in any combination.

  `-` The result of the conversion is left justified, and the right is padded with blanks. If you do not use this flag, the result is right justified, and padded on the left.

  `+` The result of a signed conversion (as determined by *type*) will always begin with a plus or minus sign. (If you do not use this flag, positive values do not begin with a plus sign.)

  `" " (space)`
  If the first character of a signed conversion specification is not a sign, or if a signed conversion results in no char-

acters, the result will begin with a space. If the space ( ) flag and the plus (+) flag both appear, the space flag is ignored.

`0` If the *type* character is `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, or `G`: leading zeroes, are used to pad the field width (following any indication of sign or base); no spaces are used for padding. If the zero (`0`) and minus (`-`) flags both appear, the zero (`0`) flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision *prec* is specified, the zero (`0`) flag is ignored. Note that `0` is interpreted as a flag, not as the beginning of a field width.

`#` The result is to be converted to an alternative form, according to the next character:

`0` increases precision to force the first digit of the result to be a zero.

`x` a non-zero result will have a `0x` prefix.

`X` a non-zero result will have a `0X` prefix.

`e, E or f` The result will always contain a decimal point even if no digits follow the point. (Normally, a decimal point appears only if a digit follows it.) Trailing zeroes are removed.

`g or G` same as `e` or `E`, but trailing zeroes are not removed.

`all others`
undefined.

- *width*

  width is an optional minimum field width. You can either specify it directly as a decimal integer, or indirectly by using instead an asterisk (`*`), in which case an `int` argument is used as the field width. Negative field widths are not supported; if you attempt to specify a negative field width, it is interpreted as a minus (`-`) flag followed by a positive field width.

- *prec*

  an optional field; if present, it is introduced with ‘`.`’ (a period). This field gives the maximum number of characters to print in a conversion; the minimum number of digits of an integer to print, for conversions with *type* `d`, `i`, `o`, `u`, `x`, and `X`; the maximum number of significant digits, for the `g` and `G` conversions; or the number of digits to print after the decimal point, for `e`, `E`, and `f` conversions. You can specify the precision either directly as a decimal integer or

indirectly by using an asterisk (`*`), in which case an `int` argument is used as the precision. Supplying a negative precision is equivalent to omitting the precision. If only a period is specified the precision is zero. If a precision appears with any other conversion *type* than those listed here, the behavior is undefined.

- *size*

  `h`, `l`, and `L` are optional size characters which override the default way that `printf` interprets the data type of the corresponding argument. `h` forces the following `d`, `i`, `o`, `u`, `x` or `X` conversion *type* to apply to a `short` or `unsigned short`. `h` also forces a following `n` *type* to apply to a pointer to a `short`. Similarily, an `l` forces the following `d`, `i`, `o`, `u`, `x` or `X` conversion *type* to apply to a `long` or `unsigned long`. `l` also forces a following `n` *type* to apply to a pointer to a `long`. If an `h` or an `l` appears with another conversion specifier, the behavior is undefined. `L` forces a following `e`, `E`, `f`, `g` or `G` conversion *type* to apply to a `long double` argument. If `L` appears with any other conversion *type*, the behavior is undefined.

- *type*

  type specifies what kind of conversion `printf` performs. Here is a table of these:

  `%` prints the percent character (`%`)

  `c` prints *arg* as single character

  `s` prints characters until precision is reached or a null terminator is encountered; takes a string pointer

  `d` prints a signed decimal integer; takes an `int` (same as `i`)

  `i` prints a signed decimal integer; takes an `int` (same as `d`)

  `o` prints a signed octal integer; takes an `int`

  `u` prints an unsigned decimal integer; takes an `int`

  `x` prints an unsigned hexadecimal integer (using `abcdef` as digits beyond `9`); takes an `int`

  `X` prints an unsigned hexadecimal integer (using `ABCDEF` as digits beyond `9`); takes an `int`

  `f` prints a signed value of the form `[-]9999.9999`; takes a floating point number

  `e` prints a signed value of the form `[-]9.9999e[+|-]999`; takes a floating point number

`E` prints the same way as `e`, but using `E` to introduce the exponent; takes a floating point number

`g` prints a signed value in either `f` or `e` form, based on given value and precision—trailing zeros and the decimal point are printed only if necessary; takes a floating point number

`G` prints the same way as `g`, but using `E` for the exponent if an exponent is needed; takes a floating point number

`n` stores (in the same object) a count of the characters written; takes a pointer to `int`

`p` prints a pointer in an implementation-defined format. This implementation treats the pointer as an `unsigned long` (same as `Lu`).

Returns

`sprintf` returns the number of bytes in the output string, save that the concluding `NULL` is not counted. `printf` and `fprintf` return the number of characters transmitted. If an error occurs, `printf` and `fprintf` return `EOF`. No error returns occur for `sprintf`.

Portability

The ANSI C standard specifies that implementations must support at least formatted output of up to 509 characters.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.36 `scanf`, `fscanf`, `sscanf`—scan and format input

Synopsis

```
#include <stdio.h>

int scanf(const char *format [, arg, ...]);
int fscanf(FILE *fd, const char *format [, arg, ...]);
int sscanf(const char *str, const char *format
    [, arg, ...]);
```

Description

`scanf` scans a series of input fields from standard input, one character at a time. Each field is interpreted according to a format specifier passed to `scanf` in the format string at `*`*`format`*. `scanf` stores the interpreted input from each field at the address passed to it as the corresponding argument following *`format`*. You must supply the same number of format specifiers and address arguments as there are input fields.

There must be sufficient address arguments for the given format specifiers; if not the results are unpredictable and likely disasterous. Excess address arguments are merely ignored.

`scanf` often produces unexpected results if the input diverges from an expected pattern. Since the combination of `gets` or `fgets` followed by `sscanf` is safe and easy, that is the preferred way to be certain that a program is synchronized with input at the end of a line.

`fscanf` and `sscanf` are identical to `scanf`, other than the source of input: `fscanf` reads from a file, and `sscanf` from a string.

The string at `*`*`format`* is a character sequence composed of zero or more directives. Directives are composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters are blank ( ), tab (`\t`), or newline (`\n`). When `scanf` encounters a whitespace character in the format string it will read (but not store) all consecutive whitespace characters up to the next non-whitespace character in the input.

Non-whitespace characters are all other ASCII characters except the percent sign (`%`). When `scanf` encounters a non-whitespace character in the format string it will read, but not store a matching non-whitespace character.

Format specifications tell `scanf` to read and convert characters from the input field into specific types of values, and store then in the locations specified by the address arguments.

Trailing whitespace is left unread unless explicitly matched in the format string.

The format specifiers must begin with a percent sign (%) and have the following form:

```
%[*][width][size]type
```

Each format specification begins with the percent character (%). The other fields are:

`*` an optional marker; if present, it suppresses interpretation and assignment of this input field.

width an optional maximum field width: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than *width* characters, `scanf` reads all the characters in the field, and then proceeds with the next field and its format specification.

If a whitespace or a non-convertable character occurs before *width* character are read, the characters up to that character are read, converted, and stored. Then `scanf` proceeds to the next format specification.

`size` `h`, `l`, and `L` are optional size characters which override the default way that `scanf` interprets the data type of the corresponding argument.

```
Modifier   Type(s)
   h       d, i, o, u, x     convert input to short,
                             store in short object

   h       D, I, O, U, X     no effect
           e, f, c, s, n, p

   l       d, i, o, u, x     convert input to long,
                             store in long object

   l       e, f, g           convert input to double
                             store in a double object

   l       D, I, O, U, X     no effect
           c, s, n, p

   L       d, i, o, u, x     convert to long double,
                             store in long double

   L      all others         no effect
```

type

A character to specify what kind of conversion `scanf` performs. Here is a table of the conversion characters:

`%` No conversion is done; the percent character (%) is stored.

`c` Scans one character. Corresponding *arg*: `(char *arg)`.

`s` Reads a character string into the array supplied. Corresponding *arg*: `(char arg[])`.

`[`*pattern*`]`
Reads a non-empty character string into memory starting at *arg*. This area must be large enough to accept the sequence and a terminating null character which will be added automatically. (*pattern* is discussed in the paragraph following this table). Corresponding *arg*: `(char *arg)`.

`d` Reads a decimal integer into the corresponding *arg*: `(int *arg)`.

`D` Reads a decimal integer into the corresponding *arg*: `(long *arg)`.

`o` Reads an octal integer into the corresponding *arg*: `(int *arg)`.

`O` Reads an octal integer into the corresponding *arg*: `(long *arg)`.

`u` Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned int *arg)`.

`U` Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned long *arg)`.

`x,X` Read a hexadecimal integer into the corresponding *arg*: `(int *arg)`.

`e, f, g` Read a floating point number into the corresponding *arg*: `(float *arg)`.

`E, F, G` Read a floating point number into the corresponding *arg*: `(double *arg)`.

`i` Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(int *arg)`.

`I` Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(long *arg)`.

`n` Stores the number of characters read in the corresponding *arg*: `(int *arg)`.

`p` Stores a scanned pointer. ANSI C leaves the details to each implementation; this implementation treats `%p` exactly the same as `%U`. Corresponding *arg*: `(void **arg)`.

A *pattern* of characters surrounded by square brackets can be used instead of the s type character. *pattern* is a set of characters which define a search set of possible characters making up the scanf input field. If the first character in the brackets is a caret (^), the search set is inverted to include all ASCII characters except those between the brackets. There is also a range facility which you can use as a shortcut. %[0-9] matches all decimal digits. The hyphen must not be the first or last character in the set. The character prior to the hyphen must be lexically less than the character after it.

Here are some *pattern* examples:

%[abcd]
: matches strings containing only a, b, c, and d.

%[^abcd]
: matches strings containing any characters except a, b, c, or d

%[A-DW-Z]
: matches strings containing A, B, C, D, W, X, Y, Z

%[z-a]
: matches the characters z, -, and a

Floating point numbers (for field types e, f, g, E, F, G) must correspond to the following general form:

```
[+/-] ddddd[.]ddd [E|e[+|-]ddd]
```

where objects inclosed in square brackets are optional, and ddd represents decimal, octal, or hexadecimal digits.

Returns

scanf returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If scanf attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

scanf might stop scanning a particular field before reaching the normal field end character, or may terminate entirely.

scanf stops scanning and storing the current field and moves to the next input field (if any) in any of the following situations:

- The assignment suppressing character (*) appears after the % in the format specification; the current input field is scanned but not stored.
- *width* characters have been read (*width* is a width specification, a positive decimal integer).
- The next character read cannot be converted under the the current format (for example, if a Z is read when the format is decimal).

- The next character in the input field does not appear in the search set (or does appear in the inverted search set).

When `scanf` stops scanning the current input field for one of these reasons, the next character is considered unread and used as the first character of the following input field, or the first character in a subsequent read operation on the input.

`scanf` will terminate under the following circumstances:

- The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
- The next character in the input field is `EOF`.
- The format string has been exhausted.

When the format string contains a character sequence that is not part of a format specification, the same character sequence must appear in the input; `scanf` will scan but not store the matched characters. If a conflict occurs, the first conflicting character remains in the input as if it had never been read.

Portability

`scanf` is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

3.37 `tmpfile`—create a temporary file

Synopsis

```
#include <stdio.h>
FILE *tmpfile(void);

FILE *_tmpfile_r(void *reent);
```

Description

Create a temporary file (a file which will be deleted automatically), using a name generated by `tmpnam`. The temporary file is opened with the mode `"wb+"`, permitting you to read and write anywhere in it as a binary file (without any data transformations the host system may perform for text files).

The alternate function `_tmpfile_r` is a reentrant version. The argument *reent* is a pointer to a reentrancy structure.

Returns

`tmpfile` normally returns a pointer to the temporary file. If no temporary file could be created, the result is NULL, and `errno` records the reason for failure.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `tmpfile`.

Supporting OS subroutines required: `close`, `fstat`, `getpid`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

`tmpfile` also requires the global pointer `environ`.

3.38 `tmpnam`, `tempnam`—name for a temporary file

Synopsis

```
#include <stdio.h>
char *tmpnam(char *s);
char *tempnam(char *dir, char *pfx);
char *_tmpnam_r(void *reent, char *s);
char *_tempnam_r(void *reent, char *dir, char *pfx);
```

Description

Use either of these functions to generate a name for a temporary file. The generated name is guaranteed to avoid collision with other files (for up to `TMP_MAX` calls of either function).

`tmpnam` generates file names with the value of `P_tmpdir` (defined in '`stdio.h`') as the leading directory component of the path.

You can use the `tmpnam` argument *s* to specify a suitable area of memory for the generated filename; otherwise, you can call `tmpnam(NULL)` to use an internal static buffer.

`tempnam` allows you more control over the generated filename: you can use the argument *dir* to specify the path to a directory for temporary files, and you can use the argument *pfx* to specify a prefix for the base filename.

If *dir* is `NULL`, `tempnam` will attempt to use the value of environment variable `TMPDIR` instead; if there is no such value, `tempnam` uses the value of `P_tmpdir` (defined in '`stdio.h`').

If you don't need any particular prefix to the basename of temporary files, you can pass `NULL` as the *pfx* argument to `tempnam`.

`_tmpnam_r` and `_tempnam_r` are reentrant versions of `tmpnam` and `tempnam` respectively. The extra argument *reent* is a pointer to a reentrancy structure.

Warnings

The generated filenames are suitable for temporary files, but do not in themselves make files temporary. Files with these names must still be explicitly removed when you no longer want them.

If you supply your own data area *s* for `tmpnam`, you must ensure that it has room for at least `L_tmpnam` elements of type `char`.

Returns

Both `tmpnam` and `tempnam` return a pointer to the newly generated filename.

Portability

ANSI C requires tmpnam, but does not specify the use of P_tmpdir. The System V Interface Definition (Issue 2) requires both tmpnam and tempnam.

Supporting OS subroutines required: close, fstat, getpid, isatty, lseek, open, read, sbrk, write.

The global pointer environ is also required.

3.39 `vprintf`, `vfprintf`, `vsprintf`—format argument list

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(const char *fmt, va_list list);
int vfprintf(FILE *fp, const char *fmt, va_list list);
int vsprintf(char *str, const char *fmt, va_list list);

int _vprintf_r(void *reent, const char *fmt,
    va_list list);
int _vfprintf_r(void *reent, FILE *fp, const char *fmt,
    va_list list);
int _vsprintf_r(void *reent, char *str, const char *fmt,
    va_list list);
```

Description

`vprintf`, `vfprintf`, and `vsprintf` are (respectively) variants of `printf`, `fprintf`, and `sprintf`. They differ only in allowing their caller to pass the variable argument list as a `va_list` object (initialized by `va_start`) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: `vsprintf` returns the number of bytes in the output string, save that the concluding `NULL` is not counted. `vprintf` and `vfprintf` return the number of characters transmitted. If an error occurs, `vprintf` and `vfprintf` return `EOF`. No error returns occur for `vsprintf`.

Portability

ANSI C requires all three functions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4 Strings and Memory (‘string.h’)

This chapter describes string-handling functions and functions for managing areas of memory. The corresponding declarations are in ‘string.h’.

4.1 `bcmp`—compare two memory areas

Synopsis

```
#include <string.h>
int bcmp(const char *s1, const char *s2, size_t n);
```

Description

This function compares not more than *n* characters of the object pointed to by *s1* with the object pointed to by *s2*.

This function is identical to `memcmp`.

Returns

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by *s1* is greater than, equal to or less than the object pointed to by *s2*.

Portability

`bcmp` requires no supporting OS subroutines.

4.2 bcopy—copy memory regions

Synopsis

```
#include <string.h>
void bcopy(const char *in, char  *out, size_t n);
```

Description

This function copies *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*.

This function is implemented in term of `memmove`.

Portability

`bcopy` requires no supporting OS subroutines.

4.3 `bzero`—initialize memory to zero

Synopsis

```
#include <string.h>
void bzero(char *b, size_t length);
```

Description

`bzero` initializes *`length`* bytes of memory, starting at address *`b`*, to zero.

Returns

`bzero` does not return a result.

Portability

`bzero` is in the Berkeley Software Distribution. Neither ANSI C nor the System V Interface Definition (Issue 2) require `bzero`.

`bzero` requires no supporting OS subroutines.

4.4 `index`—search for character in string

Synopsis

```
#include <string.h>
char * index(const char *string, int c);
```

Description

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

This function is identical to `strchr`.

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`index` requires no supporting OS subroutines.

4.5 `memchr`—find character in memory

Synopsis

```
#include <string.h>
void *memchr(const void *src, int c, size_t length);
```

Description

This function searches memory starting at `*`*`src`* for the character *`c`*. The search only ends with the first occurrence of *`c`*, or after *`length`* characters; in particular, `NULL` does not terminate the search.

Returns

If the character *`c`* is found within *`length`* characters of `*`*`src`*, a pointer to the character is returned. If *`c`* is not found, then `NULL` is returned.

Portability

`memchr>` is ANSI C.

`memchr` requires no supporting OS subroutines.

4.6 memcmp—compare two memory areas

Synopsis

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

This function compares not more than *n* characters of the object pointed to by *s1* with the object pointed to by *s2*.

Returns

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by *s1* is greater than, equal to or less than the object pointed to by *s2*.

Portability

`memcmp` is ANSI C.

`memcmp` requires no supporting OS subroutines.

4.7 `memcpy`—copy memory regions

Synopsis

```
#include <string.h>
void* memcpy(void *out, const void *in, size_t n);
```

Description

This function copies *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*.

If the regions overlap, the behavior is undefined.

Returns

`memcpy` returns a pointer to the first byte of the *out* region.

Portability

`memcpy` is ANSI C.

`memcpy` requires no supporting OS subroutines.

4.8 `memmove`—move possibly overlapping memory

Synopsis

```
#include <string.h>
void *memmove(void *dst, const void *src, size_t length);
```

Description

This function moves *length* characters from the block of memory starting at **src* to the memory starting at **dst*. `memmove` reproduces the characters correctly at **dst* even if the two areas overlap.

Returns

The function returns *dst* as passed.

Portability

`memmove` is ANSI C.

`memmove` requires no supporting OS subroutines.

4.9 `memset`—set an area of memory

Synopsis

```
#include <string.h>
void *memset(const void *dst, int c, size_t length);
```

Description

This function converts the argument *c* into an unsigned char and fills the first *length* characters of the array pointed to by *dst* to the value.

Returns

`memset` returns the value of *m*.

Portability

`memset` is ANSI C.

`memset` requires no supporting OS subroutines.

4.10 rindex—reverse search for character in string

Synopsis

```
#include <string.h>
char * rindex(const char *string, int c);
```

Description

This function finds the last occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

This function is identical to strrchr.

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

rindex requires no supporting OS subroutines.

4.11 `strcat`—concatenate strings

Synopsis

```
#include <string.h>
char *strcat(char *dst, const char *src);
```

Description

`strcat` appends a copy of the string pointed to by *src* (including the terminating null character) to the end of the string pointed to by *dst*. The initial character of *src* overwrites the null character at the end of *dst*.

Returns

This function returns the initial value of *dst*

Portability

`strcat` is ANSI C.

`strcat` requires no supporting OS subroutines.

4.12 `strchr`—search for character in string

Synopsis

```
#include <string.h>
char * strchr(const char *string, int c);
```

Description

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`strchr` is ANSI C.

`strchr` requires no supporting OS subroutines.

4.13 `strcmp`—character string compare

Synopsis

```
#include <string.h>
int strcmp(const char *a, const char *b);
```

Description

`strcmp` compares the string at *a* to the string at *b*.

Returns

If `*a` sorts lexicographically after `*b`, `strcmp` returns a number greater than zero. If the two strings match, `strcmp` returns zero. If `*a` sorts lexicographically before `*b`, `strcmp` returns a number less than zero.

Portability

`strcmp` is ANSI C.

`strcmp` requires no supporting OS subroutines.

4.14 `strcoll`—locale specific character string compare

Synopsis

```
#include <string.h>
int strcoll(const char *stra, const char * strb);
```

Description

`strcoll` compares the string pointed to by *stra* to the string pointed to by *strb*, using an interpretation appropriate to the current `LC_COLLATE` state.

Returns

If the first string is greater than the second string, `strcoll` returns a number greater than zero. If the two strings are equivalent, `strcoll` returns zero. If the first string is less than the second string, `strcoll` returns a number less than zero.

Portability

`strcoll` is ANSI C.

`strcoll` requires no supporting OS subroutines.

4.15 `strcpy`—copy string

Synopsis

```
#include <string.h>
char *strcpy(char *dst, const char *src);
```

Description

`strcpy` copies the string pointed to by *src* (including the terminating null character) to the array pointed to by *dst*.

Returns

This function returns the initial value of *dst*.

Portability

`strcpy` is ANSI C.

`strcpy` requires no supporting OS subroutines.

4.16 `strcspn`—count chars not in string

Synopsis

```
size_t strcspn(const char *s1, const char *s2);
```

Description

This function computes the length of the initial part of the string pointed to by *s1* which consists entirely of characters *NOT* from the string pointed to by *s2* (excluding the terminating null character).

Returns

`strcspn` returns the length of the substring found.

Portability

`strcspn` is ANSI C.

`strcspn` requires no supporting OS subroutines.

4.17 `strerror`—convert error number to string

Synopsis

```
#include <string.h>
char *strerror(int errnum);
```

Description

`strerror` converts the error number *errnum* into a string. The value of *errnum* is usually a copy of `errno`. If `errnum` is not a known error number, the result points to an empty string.

This implementation of `strerror` prints out the following strings for each of the values defined in '`errno.h`':

`E2BIG` Arg list too long

`EACCES` Permission denied

`EADV` Advertise error

`EAGAIN` No more processes

`EBADF` Bad file number

`EBADMSG` Bad message

`EBUSY` Device or resource busy

`ECHILD` No children

`ECOMM` Communication error

`EDEADLK` Deadlock

`EEXIST` File exists

`EDOM` Math argument

`EFAULT` Bad address

`EFBIG` File too large

`EIDRM` Identifier removed

`EINTR` Interrupted system call

`EINVAL` Invalid argument

`EIO` I/O error

`EISDIR` Is a directory

`ELIBACC` Cannot access a needed shared library

`ELIBBAD` Accessing a corrupted shared library

`ELIBEXEC` Cannot exec a shared library directly

`ELIBMAX` Attempting to link in more shared libraries than system limit

`ELIBSCN` `.lib` section in a.out corrupted

`EMFILE` Too many open files

`EMLINK` Too many links

`EMULTIHOP`
Multihop attempted

`ENFILE` File table overflow

`ENODEV` No such device

`ENOENT` No such file or directory

`ENOEXEC` Exec format error

`ENOLCK` No lock

`ENOLINK` Virtual circuit is gone

`ENOMEM` Not enough space

`ENOMSG` No message of desired type

`ENONET` Machine is not on the network

`ENOPKG` No package

`ENOSPC` No space left on device

`ENOSR` No stream resources

`ENOSTR` Not a stream

`ENOTBLK` Block device required

`ENOTDIR` Not a directory

`ENOTTY` Not a character device

`ENXIO` No such device or address

`EPERM` Not owner

`EPIPE` Broken pipe

`EPROTO` Protocol error

`ERANGE` Result too large

`EREMOTE` Resource is remote

`EROFS` Read-only file system

`ESPIPE` Illegal seek

`ESRCH` No such process

`ESRMNT` Srmount error

`ETIME` Stream ioctl timeout

`ETXTBSY` Text file busy

`EXDEV` Cross-device link

Returns

This function returns a pointer to a string. Your application must not modify that string.

Portability

ANSI C requires `strerror`, but does not specify the strings used for each error number.

Although this implementation of `strerror` is reentrant, ANSI C declares that subsequent calls to `strerror` may overwrite the result string; therefore portable code cannot depend on the reentrancy of this subroutine.

`strerror` requires no supporting OS subroutines.

4.18 strlen—character string length

Synopsis

```
#include <string.h>
size_t strlen(const char *str);
```

Description

The `strlen` function works out the length of the string starting at `*str` by counting chararacters until it reaches a `NULL` character.

Returns

`strlen` returns the character count.

Portability

`strlen` is ANSI C.

`strlen` requires no supporting OS subroutines.

4.19 `strncat`—concatenate strings

Synopsis

```
#include <string.h>
char *strncat(char *dst, const char *src, size_t length);
```

Description

`strncat` appends not more than *length* characters from the string pointed to by *src* (including the terminating null character) to the end of the string pointed to by *dst*. The initial character of *src* overwrites the null character at the end of *dst*. A terminating null character is always appended to the result

Warnings

Note that a null is always appended, so that if the copy is limited by the *length* argument, the number of characters appended to *dst* is `n + 1`.

Returns

This function returns the initial value of *dst*

Portability

`strncat` is ANSI C.

`strncat` requires no supporting OS subroutines.

4.20 `strncmp`—character string compare

Synopsis

```
#include <string.h>
int strncmp(const char *a, const char * b, size_t length);
```

Description

`strncmp` compares up to *length* characters from the string at *a* to the string at *b*.

Returns

If `*a` sorts lexicographically after `*b`, `strncmp` returns a number greater than zero. If the two strings are equivalent, `strncmp` returns zero. If `*a` sorts lexicographically before `*b`, `strncmp` returns a number less than zero.

Portability

`strncmp` is ANSI C.

`strncmp` requires no supporting OS subroutines.

4.21 `strncpy`—counted copy string

Synopsis

```
#include <string.h>
char *strncpy(char *dst, const char *src, size_t length);
```

Description

`strncpy` copies not more than *`length`* characters from the the string pointed to by *`src`* (including the terminating null character) to the array pointed to by *`dst`*. If the string pointed to by *`src`* is shorter than *`length`* characters, null characters are appended to the destination array until a total of *`length`* characters have been written.

Returns

This function returns the initial value of *`dst`*.

Portability

`strncpy` is ANSI C.

`strncpy` requires no supporting OS subroutines.

4.22 `strpbrk`—find chars in string

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

This function locates the first occurence in the string pointed to by *s1* of any character in string pointed to by *s2* (excluding the terminating null character).

Returns

`strpbrk` returns a pointer to the character found in *s1*, or a null pointer if no character from *s2* occurs in *s1*.

Portability

`strpbrk` requires no supporting OS subroutines.

4.23 `strrchr`—reverse search for character in string

Synopsis

```
#include <string.h>
char * strrchr(const char *string, int c);
```

Description

This function finds the last occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`strrchr` is ANSI C.

`strrchr` requires no supporting OS subroutines.

4.24 strspn—find initial match

Synopsis

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description

This function computes the length of the initial segment of the string pointed to by *s1* which consists entirely of characters from the string pointed to by *s2* (excluding the terminating null character).

Returns

`strspn` returns the length of the segment found.

Portability

`strspn` is ANSI C.

`strspn` requires no supporting OS subroutines.

4.25 `strstr`—find string segment

Synopsis

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description

Locates the first occurence in the string pointed to by *s1* of the sequence of characters in the string pointed to by *s2* (excluding the terminating null character).

Returns

Returns a pointer to the located string segment, or a null pointer if the string *s2* is not found. If *s2* points to a string with zero length, the *s1* is returned.

Portability

`strstr` is ANSI C.

`strstr` requires no supporting OS subroutines.

4.26 `strtok`—get next token from a string

Synopsis

```
#include <string.h>
char *strtok(char *source, const char *delimiters)

char *_strtok_r(void *reent,
    const char *source, const char *delimiters)
```

Description

A series of calls to `strtok` break the string starting at `*source` into a sequence of tokens. The tokens are delimited from one another by characters from the string at `*delimiters`, at the outset. The first call to `strtok` normally has a string address as the first argument; subsequent calls can use `NULL` as the first argument, to continue searching the same string. You can continue searching a single string with different delimiters by using a different delimiter string on each call.

`strtok` begins by searching for any character not in the *delimiters* string: the first such character is the beginning of a token (and its address will be the result of the `strtok` call). `strtok` then continues searching until it finds another delimiter character; it replaces that character by `NULL` and returns. (If `strtok` comes to the end of the `*source` string without finding any more delimiters, the entire remainder of the string is treated as the next token). `strtok` starts its search at `*source`, unless you pass `NULL` as the first argument; if *source* is `NULL`, `strtok` continues searching from the end of the last search. Exploiting the `NULL` first argument leads to non-reentrant code. You can easily circumvent this problem by saving the last delimiter address in your application, and always using it to pass a non-null *source* argument.

`_strtok_r` performs the same function as `strtok`, but is reentrant. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`strtok` returns a pointer to the next token, or `NULL` if no more tokens can be found.

Portability

`strtok` is ANSI C.

`strtok` requires no supporting OS subroutines.

4.27 `strxfrm`—transform string

Synopsis

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Description

This function transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp` function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, correspoinding to the result of a `strcoll` function applied to the same two original strings.

No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* may be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

With a C locale, this function just copies.

Returns

The `strxfrm` function returns the length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

Portability

`strxfrm` is ANSI C.

`strxfrm` requires no supporting OS subroutines.

5 Signal Handling (‘signal.h’)

A *signal* is an event that interrupts the normal flow of control in your program. Your operating environment normally defines the full set of signals available (see ‘sys/signal.h’), as well as the default means of dealing with them—typically, either printing an error message and aborting your program, or ignoring the signal.

All systems support at least the following signals:

SIGABRT — Abnormal termination of a program; raised by the <<abort>> function.

SIGFPE — A domain error in arithmetic, such as overflow, or division by zero.

SIGILL — Attempt to execute as a function data that is not executable.

SIGINT — Interrupt; an interactive attention signal.

SIGSEGV — An attempt to access a memory location that is not available.

SIGTERM — A request that your program end execution.

Two functions are available for dealing with asynchronous signals—one to allow your program to send signals to itself (this is called *raising* a signal), and one to specify subroutines (called *handlers* to handle particular signals that you anticipate may occur—whether raised by your own program or the operating environment.

To support these functions, ‘signal.h’ defines three macros:

SIG_DFL — Used with the signal function in place of a pointer to a handler subroutine, to select the operating environment’s default handling of a signal.

SIG_IGN — Used with the signal function in place of a pointer to a handler, to ignore a particular signal.

SIG_ERR — Returned by the signal function in place of a pointer to a handler, to indicate that your request to set up a handler could not be honored for some reason.

‘signal.h’ also defines an integral type, sig_atomic_t. This type is not used in any function declarations; it exists only to allow your signal handlers to declare a static storage location where they may store a signal value. (Static storage is not otherwise reliable from signal handlers.)

5.1 `raise`—send a signal

Synopsis

```
#include <signal.h>
int raise(int sig);

int _raise_r(void *reent, int sig);
```

Description

Send the signal *sig* (one of the macros from 'sys/signal.h'). This interrupts your program's normal flow of execution, and allows a signal handler (if you've defined one, using `signal`) to take control.

The alternate function `_raise_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The result is `0` if *sig* was successfully raised, `1` otherwise. However, the return value (since it depends on the normal flow of execution) may not be visible, unless the signal handler for *sig* terminates with a `return` or unless `SIG_IGN` is in effect for this signal.

Portability

ANSI C requires `raise`, but allows the full set of signal numbers to vary from one implementation to another.

Required OS subroutines: `getpid`, `kill`.

5.2 signal—specify handler subroutine for a signal

Synopsis

```
#include <signal.h>
void ( * signal(int sig, void(*func)(int)) )(int);

void ( * _signal_r(void *reent,
    int sig, void(*func)(int)) )(int);

int raise (int sig);

int _raise_r (void *reent, int sig);
```

Description

signal, raise provide a simple signal/raise implementation for embedded targets.

signal allows you to request changed treatment for a particular signal *sig*. You can use one of the predefined macros SIG_DFL (select system default handling) or SIG_IGN (ignore this signal) as the value of *func*; otherwise, *func* is a function pointer that identifies a subroutine in your program as the handler for this signal.

Some of the execution environment for signal handlers is unpredictable; notably, the only library function required to work correctly from within a signal handler is signal itself, and only when used to redefine the handler for the current signal value.

Static storage is likewise unreliable for signal handlers, with one exception: if you declare a static storage location as ‘volatile sig_atomic_t’, then you may use that location in a signal handler to store signal values.

If your signal handler terminates using return (or implicit return), your program's execution continues at the point where it was when the signal was raised (whether by your program itself, or by an external event). Signal handlers can also use functions such as exit and abort to avoid returning.

raise sends the signal sig to the executing program. It returns zero if successful, non-zero if unsuccessful.

The alternate functions _signal_r, _raise_r are the reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If your request for a signal handler cannot be honored, the result is SIG_ERR; a specific error number is also recorded in errno.

Otherwise, the result is the previous handler (a function pointer or one of the predefined macros).

Portability

ANSI C requires `raise`, `signal`.

No supporting OS subroutines are required to link with `signal`, but it will not have any useful effects, except for software generated signals, without an operating system that can actually raise exceptions.

6 Time Functions (‘time.h’)

This chapter groups functions used either for reporting on time (elapsed, current, or compute time) or to perform calculations based on time.

The header file ‘time.h’ defines three types. `clock_t` and `time_t` are both used for representations of time particularly suitable for arithmetic. (In this implementation, quantities of type `clock_t` have the highest resolution possible on your machine, and quantities of type `time_t` resolve to seconds.) `size_t` is also defined if necessary for quantities representing sizes.

‘time.h’ also defines the structure `tm` for the traditional representation of Gregorian calendar time as a series of numbers, with the following fields:

`tm_sec` Seconds.

`tm_min` Minutes.

`tm_hour` Hours.

`tm_mday` Day.

`tm_mon` Month.

`tm_year` Year (since 1900).

`tm_wday` Day of week: the number of days since Sunday.

`tm_yday` Number of days elapsed since last January 1.

`tm_isdst` Daylight Savings Time flag: positive means DST in effect, zero means DST not in effect, negative means no information about DST is available.

6.1 `asctime`—format time as string

Synopsis

```
#include <time.h>
char *asctime(const struct tm *timp);

#include <time.h>
char *_asctime_r(const struct tm *timp, void *reent);
```

Description

Format the time value at *timp* into a string of the form

```
Wed Jun 15 11:38:07 1988\n\0
```

The string is generated in a static buffer; each call to `asctime` overwrites the string generated by previous calls.

`_asctime_r` provides the same function as `asctime`, but is reentrant. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

A pointer to the string containing a formatted timestamp.

Portability

ANSI C requires `asctime`.

`asctime` requires no supporting OS subroutines.

6.2 clock—cumulative processor time

Synopsis

```
#include <time.h>
clock_t clock(void);
```

Description

Calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro `CLOCKS_PER_SEC`.

Returns

The amount of processor time used so far by your program, in units defined by the machine-dependent macro `CLOCKS_PER_SEC`. If no measurement is available, the result is `-1`.

Portability

ANSI C requires `clock` and `CLOCKS_PER_SEC`.

Supporting OS subroutine required: `times`.

6.3 `ctime`—convert time to local and format as string

Synopsis

```
#include <time.h>
char *ctime(time_t timp);
```

Description

Convert the time value at *`timp`* to local time (like `localtime`) and format it into a string of the form

```
Wed Jun 15 11:38:07 1988\n\0
```

(like `asctime`).

Returns

A pointer to the string containing a formatted timestamp.

Portability

ANSI C requires `ctime`.

`ctime` requires no supporting OS subroutines.

6.4 `difftime`—subtract two times

Synopsis

```
#include <time.h>
double difftime(time_t tim1, time_t tim2);
```

Description

Subtracts the two times in the arguments: ‘*tim1* - *tim2*’.

Returns

The difference (in seconds) between *tim2* and *tim1*, as a `double`.

Portability

ANSI C requires `difftime`, and defines its result to be in seconds in all implementations.

`difftime` requires no supporting OS subroutines.

6.5 `gmtime`—convert time to UTC traditional form

Synopsis

```
#include <time.h>
struct tm *gmtime(const time_t *timep
```

Description

`gmtime` assumes the time at *`timep`* represents a local time. `gmtime` converts it to UTC (Universal Coordinated Time, also known in some countries as GMT, Greenwich Mean time), then converts the representation from the arithmetic representation to the traditional representation defined by `struct tm`.

`gmtime` constructs the traditional time representation in static storage; each call to `gmtime` or `localtime` will overwrite the information generated by previous calls to either function.

Returns

A pointer to the traditional time representation (`struct tm`).

Portability

ANSI C requires `gmtime`.

`gmtime` requires no supporting OS subroutines.

6.6 `localtime`—convert time to local representation

Synopsis

```
#include <time.h>
struct tm *localtime(time_t *timep);
```

Description

`localtime` converts the time at *timep* into local time, then converts its representation from the arithmetic representation to the traditional representation defined by `struct tm`.

`localtime` constructs the traditional time representation in static storage; each call to `gmtime` or `localtime` will overwrite the information generated by previous calls to either function.

`mktime` is the inverse of `localtime`.

Returns

A pointer to the traditional time representation (`struct tm`).

Portability

ANSI C requires `localtime`.

`localtime` requires no supporting OS subroutines.

6.7 `mktime`—convert time to arithmetic representation

Synopsis

```
#include <time.h>
time_t mktime(struct tm *timp);
```

Description

`mktime` assumes the time at *timp* is a local time, and converts its representation from the traditional representation defined by `struct tm` into a representation suitable for arithmetic.

`localtime` is the inverse of `mktime`.

Returns

If the contents of the structure at *timp* do not form a valid calendar time representation, the result is `-1`. Otherwise, the result is the time, converted to a `time_t` value.

Portability

ANSI C requires `mktime`.

`mktime` requires no supporting OS subroutines.

6.8 `strftime`—flexible calendar time formatter

Synopsis

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
    const char *format, const struct tm *timp);
```

Description

`strftime` converts a `struct tm` representation of the time (at *timp*) into a string, starting at *s* and occupying no more than *maxsize* characters.

You control the format of the output using the string at *format*. **format* can contain two kinds of specifications: text to be copied literally into the formatted string, and time conversion specifications. Time conversion specifications are two-character sequences beginning with '`%`' (use '`%%`' to include a percent sign in the output). Each defined conversion specification selects a field of calendar time data from **timp*, and converts it to a string in one of the following ways:

`%a` An abbreviation for the day of the week.

`%A` The full name for the day of the week.

`%b` An abbreviation for the month name.

`%B` The full name of the month.

`%c` A string representing the complete date and time, in the form

```
Mon Apr 01 13:13:13 1992
```

`%d` The day of the month, formatted with two digits.

`%H` The hour (on a 24-hour clock), formatted with two digits.

`%I` The hour (on a 12-hour clock), formatted with two digits.

`%j` The count of days in the year, formatted with three digits (from '`001`' to '`366`').

`%m` The month number, formatted with two digits.

`%M` The minute, formatted with two digits.

`%p` Either '`AM`' or '`PM`' as appropriate.

`%S` The second, formatted with two digits.

`%U` The week number, formatted with two digits (from '`00`' to '`53`'; week number 1 is taken as beginning with the first Sunday in a year). See also `%W`.

`%w` A single digit representing the day of the week: Sunday is day `0`.

`%W` Another version of the week number: like '`%U`', but counting week 1 as beginning with the first Monday in a year.

o `%x` A string representing the complete date, in a format like

```
Mon Apr 01 1992
```

`%X` A string representing the full time of day (hours, minutes, and seconds), in a format like

```
13:13:13
```

`%y` The last two digits of the year.

`%Y` The full year, formatted with four digits to include the century.

`%Z` Defined by ANSI C as eliciting the time zone if available; it is not available in this implementation (which accepts '`%Z`' but generates no output for it).

`%%` A single character, '`%`'.

Returns

When the formatted time takes up no more than *maxsize* characters, the result is the length of the formatted string. Otherwise, if the formatting operation was abandoned due to lack of room, the result is `0`, and the string starting at *s* corresponds to just those parts of `*`*format* that could be completely filled in within the *maxsize* limit.

Portability

ANSI C requires `strftime`, but does not specify the contents of `*`*s* when the formatted string would require more than *maxsize* characters.

`strftime` requires no supporting OS subroutines.

6.9 time—get current calendar time (as single number)

Synopsis

```
#include <time.h>
time_t time(time_t *t);
```

Description

time looks up the best available representation of the current time and returns it, encoded as a time_t. It stores the same value at *t* unless the argument is NULL.

Returns

A -1 result means the current time is not available; otherwise the result represents the current time.

Portability

ANSI C requires time.

Supporting OS subroutine required: Some implementations require gettimeofday.

7 Locale ('locale.h')

A *locale* is the name for a collection of parameters (affecting collating sequences and formatting conventions) that may be different depending on location or culture. The "C" locale is the only one defined in the ANSI C standard.

This is a minimal implementation, supporting only the required ``C'' value for locale; strings representing other locales are not honored. ("" is also accepted; it represents the default locale for an implementation, here equivalent to ``C''.

'locale.h' defines the structure lconv to collect the information on a locale, with the following fields:

char *decimal_point
: The decimal point character used to format "ordinary" numbers (all numbers except those referring to amounts of money). ``.'' in the C locale.

char *thousands_sep
: The character (if any) used to separate groups of digits, when formatting ordinary numbers. ``'' in the C locale.

char *grouping
: Specifications for how many digits to group (if any grouping is done at all) when formatting ordinary numbers. The *numeric value* of each character in the string represents the number of digits for the next group, and a value of 0 (that is, the string's trailing NULL) means to continue grouping digits using the last value specified. Use CHAR_MAX to indicate that no further grouping is desired. ``'' in the C locale.

char *int_curr_symbol
: The international currency symbol (first three characters), if any, and the character used to separate it from numbers. ``'' in the C locale.

char *currency_symbol
: The local currency symbol, if any. ``'' in the C locale.

char *mon_decimal_point
: The symbol used to delimit fractions in amounts of money. ``'' in the C locale.

char *mon_thousands_sep
: Similar to thousands_sep, but used for amounts of money. ``'' in the C locale.

`char *mon_grouping`
: Similar to `grouping`, but used for amounts of money. `''''` in the C locale.

`char *positive_sign`
: A string to flag positive amounts of money when formatting. `''''` in the C locale.

`char *negative_sign`
: A string to flag negative amounts of money when formatting. `''''` in the C locale.

`char int_frac_digits`
: The number of digits to display when formatting amounts of money to international conventions. `CHAR_MAX` (the largest number representable as a `char`) in the C locale.

`char frac_digits`
: The number of digits to display when formatting amounts of money to local conventions. `CHAR_MAX` in the C locale.

`char p_cs_precedes`
: `1` indicates the local currency symbol is used before a *positive or zero* formatted amount of money; `0` indicates the currency symbol is placed after the formatted number. `CHAR_MAX` in the C locale.

`char p_sep_by_space`
: `1` indicates the local currency symbol must be separated from *positive or zero* numbers by a space; `0` indicates that it is immediately adjacent to numbers. `CHAR_MAX` in the C locale.

`char n_cs_precedes`
: `1` indicates the local currency symbol is used before a *negative* formatted amount of money; `0` indicates the currency symbol is placed after the formatted number. `CHAR_MAX` in the C locale.

`char n_sep_by_space`
: `1` indicates the local currency symbol must be separated from *negative* numbers by a space; `0` indicates that it is immediately adjacent to numbers. `CHAR_MAX` in the C locale.

`char p_sign_posn`
: Controls the position of the *positive* sign for numbers representing money. `0` means parentheses surround the number; `1` means the sign is placed before both the number and the currency symbol; `2` means the sign is placed after both the number and the currency symbol; `3` means the sign is placed

just before the currency symbol; and 4 means the sign is placed just after the currency symbol. CHAR_MAX in the C locale.

char n_sign_posn
: Controls the position of the *negative* sign for numbers representing money, using the same rules as p_sign_posn. CHAR_MAX in the C locale.

7.1 setlocale, localeconv—select or query locale

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char *locale);
lconv *localeconv(void);

char *_setlocale_r(void *reent,
    int category, const char *locale);
lconv *_localeconv_r(void *reent);
```

Description

`setlocale` is the facility defined by ANSI C to condition the execution environment for international collating and formatting information; `localeconv` reports on the settings of the current locale.

This is a minimal implementation, supporting only the required `` ``C'' `` value for *locale*; strings representing other locales are not honored. (`` ``'' `` is also accepted; it represents the default locale for an implementation, here equivalent to `` ``C'' ``.)

If you use `NULL` as the *locale* argument, `setlocale` returns a pointer to the string representing the current locale (always `` ``C'' `` in this implementation). The acceptable values for *category* are defined in '`locale.h`' as macros beginning with `"LC_"`, but this implementation does not check the values you pass in the *category* argument.

`localeconv` returns a pointer to a structure (also defined in '`locale.h`') describing the locale-specific conventions currently in effect.

`_localeconv_r` and `_setlocale_r` are reentrant versions of `localeconv` and `setlocale` respectively. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`setlocale` returns either a pointer to a string naming the locale currently in effect (always `` ``C'' `` for this implementation), or, if the locale request cannot be honored, `NULL`.

`localeconv` returns a pointer to a structure of type `lconv`, which describes the formatting and collating conventions in effect (in this implementation, always those of the C locale).

Portability

ANSI C requires `setlocale`, but the only locale required across all implementations is the C locale.

No supporting OS subroutines are required.

8 Reentrancy

Reentrancy is a characteristic of library functions which allows multiple processes to use the same address space with assurance that the values stored in those spaces will remain constant between calls. Cygnus's implementation of the library functions ensures that whenever possible, these library functions are reentrant. However, there are some functions that can not be trivially made reentrant. Hooks have been provided to allow you to use these functions in a fully reentrant fashion.

These hooks use the structure `_reent` defined in '`reent.h`'. A variable defined as '`struct _reent`' is called a *reentrancy structure*. All functions which must manipulate global information are available in two versions. The first version has the usual name, and uses a single global instance of the reentrancy structure. The second has a different name, normally formed by prepending '`_`' and appending '`_r`', and takes a pointer to the particular reentrancy structure to use.

For example, the function `fopen` takes two arguments, *`file`* and *`mode`*, and uses the global reentrancy structure. The function `_fopen_r` takes the arguments, *`struct_reent`*, which is a pointer to an instance of the reentrancy structure, *`file`* and *`mode`*.

Each function which uses the global reentrancy structure uses the global variable `_impure_ptr`, which points to a reentrancy structure.

This means that you have two ways to achieve reentrancy. Both require that each thread of execution control initialize a unique global variable of type '`struct _reent`':

1. Use the reentrant versions of the library functions, after initializing a global reentrancy structure for each process. Use the pointer to this structure as the extra argument for all library functions.
2. Ensure that each thread of execution control has a pointer to its own unique reentrancy structure in the global variable `_impure_ptr`, and call the standard library subroutines.

The following functions are provided in both reentrant and non-reentrant versions.

Equivalent for errno variable:

```
_errno_r
```

Locale functions:

```
_localeconv_r  _setlocale_r
```

Equivalents for stdio variables:

```
_stdin_r        _stdout_r       _stderr_r
```

Stdio functions:

_fdopen_r	_mkstemp_r	_remove_r
_fopen_r	_mktemp_r	_rename_r
_getchar_r	_perror_r	_tempnam_r
_gets_r	_putchar_r	_tmpnam_r
_iprintf_r	_puts_r	_tmpfile_r

Signal functions:

_raise_r	_signal_r

Stdlib functions:

_dtoa_r	_realloc_r	_strtoul_r
_free_r	_srand_r	_system_r
_malloc_r	_strtod_r	
_rand_r	_strtol_r	

String functions:

_strtok_r

System functions:

_close_r	_lseek_r	_stat_r
_fork_r	_open_r	_unlink_r
_fstat_r	_read_r	_wait_r
_link_r	_sbrk_r	_write_r

Time function:

_asctime_r

9 System Calls

The C subroutine library depends on a handful of subroutine calls for operating system services. If you use the C library on a system that complies with the POSIX.1 standard (also known as IEEE 1003.1), most of these subroutines are supplied with your operating system.

If some of these subroutines are not provided with your system—in the extreme case, if you are developing software for a "bare board" system, without an OS—you will at least need to provide do-nothing stubs (or subroutines with minimal functionality) to allow your programs to link with the subroutines in `libc.a`.

9.1 Definitions for OS interface

This is the complete set of system definitions (primarily subroutines) required; the examples shown implement the minimal functionality required to allow `libc` to link, and fail gracefully where OS services are not available.

Graceful failure is permitted by returning an error code. A minor complication arises here: the C library must be compatible with development environments that supply fully functional versions of these subroutines. Such environments usually return error codes in a global `errno`. However, the Cygnus C library provides a *macro* definition for `errno` in the header file '`errno.h`', as part of its support for reentrant routines (see Chapter 8 "Reentrancy," page 149).

The bridge between these two interpretations of `errno` is straightforward: the C library routines with OS interface calls capture the `errno` values returned globally, and record them in the appropriate field of the reentrancy structure (so that you can query them using the `errno` macro from '`errno.h`').

This mechanism becomes visible when you write stub routines for OS interfaces. You must include '`errno.h`', then disable the macro, like this:

```
#include <errno.h>
#undef errno
extern int errno;
```

The examples in this chapter include this treatment of `errno`.

`_exit` Exit a program without cleaning up files. If your system doesn't provide this, it is best to avoid linking with subroutines that require it (`exit`, `system`).

`close` Close a file. Minimal implementation:

```
int close(int file){
    return -1;
```

```
}
```

`environ` A pointer to a list of environment variables and their values. For a minimal environment, this empty list is adequate:

```
char *__env[1] = { 0 };
char **environ = __env;
```

`execve` Transfer control to a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int execve(char *name, char **argv, char **env){
  errno=ENOMEM;
  return -1;
}
```

`fork` Create a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int fork() {
  errno=EAGAIN;
  return -1;
}
```

`fstat` Status of an open file. For consistency with other minimal implementations in these examples, all files are regarded as character special devices. The '`sys/stat.h`' header file required is distributed in the '`include`' subdirectory for this C library.

```
#include <sys/stat.h>
int fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

`getpid` Process-ID; this is sometimes used to generate strings unlikely to conflict with other processes. Minimal implementation, for a system without processes:

```
int getpid() {
  return 1;
}
```

`isatty` Query whether output stream is a terminal. For consistency with the other minimal implementations, which only support output to `stdout`, this minimal implementation is suggested:

```
int isatty(int file){
   return 1;
}
```

`kill` Send a signal. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int kill(int pid, int sig){
  errno=EINVAL;
  return(-1);
}
```

`link` Establish a new name for an existing file. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int link(char *old, char *new){
  errno=EMLINK;
  return -1;
}
```

`lseek` Set position in a file. Minimal implementation:

```
int lseek(int file, int ptr, int dir){
    return 0;
}
```

`read` Read from a file. Minimal implementation:

```
int read(int file, char *ptr, int len){
    return 0;
}
```

`sbrk` Increase program data space. As `malloc` and related functions depend on this, it is useful to have a working implementation. The following suffices for a standalone system; it exploits the symbol `end` automatically defined by the GNU linker.

```
caddr_t sbrk(int incr){
  extern char end; /* Defined by the linker */
  static char *heap_end;
  char *prev_heap_end;

  if (heap_end == 0) {
    heap_end = &end;
  }
  prev_heap_end = heap_end;
  heap_end += incr;
  return (caddr_t) prev_heap_end;
}
```

`stat` Status of a file (by name). Minimal implementation:

```
int stat(char *file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

`times` Timing information for current process. Minimal implementation:

```
int times(struct tms *buf){
  return -1;
}
```

`unlink` Remove a file's directory entry. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int unlink(char *name){
  errno=ENOENT;
  return -1;
}
```

`wait` Wait for a child process. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int wait(int *status) {
  errno=ECHILD;
  return -1;
}
```

`write` Write a character to a file. '`libc`' subroutines will use this system routine for output to all files, *including* `stdout`—so if you need to generate any output, for example to a serial

port for debugging, you should make your minimal `write` capable of doing this. The following minimal implementation is an incomplete example; it relies on a `writechar` subroutine (not shown; typically, you must write this in assembler from examples provided by your hardware manufacturer) to actually perform the output.

```
int write(int file, char *ptr, int len){
    int todo;

    for (todo = 0; todo < len; todo++) {
        writechar(*ptr++);
    }
    return len;
}
```

9.2 Reentrant covers for OS subroutines

Since the system subroutines are used by other library routines that require reentrancy, ‘libc.a’ provides cover routines (for example, the reentrant version of fork is _fork_r). These cover routines are consistent with the other reentrant subroutines in this library, and achieve reentrancy by using a reserved global data block (see Chapter 8 “Reentrancy,” page 149).

_open_r
: A reentrant version of open. It takes a pointer to the global data block, which holds errno.

```
int _open_r(void *reent,
    const char *file, int flags, int mode);
```

_close_r
: A reentrant version of close. It takes a pointer to the global data block, which holds errno.

```
int _close_r(void *reent, int fd);
```

_lseek_r
: A reentrant version of lseek. It takes a pointer to the global data block, which holds errno.

```
off_t _lseek_r(void *reent,
    int fd, off_t pos, int whence);
```

_read_r
: A reentrant version of read. It takes a pointer to the global data block, which holds errno.

```
long _read_r(void *reent,
    int fd, void *buf, size_t cnt);
```

_write_r
: A reentrant version of write. It takes a pointer to the global data block, which holds errno.

```
long _write_r(void *reent,
    int fd, const void *buf, size_t cnt);
```

_fork_r
: A reentrant version of fork. It takes a pointer to the global data block, which holds errno.

```
int _fork_r(void *reent);
```

_wait_r
: A reentrant version of wait. It takes a pointer to the global data block, which holds errno.

```
int _wait_r(void *reent, int *status);
```

_stat_r
: A reentrant version of stat. It takes a pointer to the global data block, which holds errno.

```
int _stat_r(void *reent,
    const char *file, struct stat *pstat);
```

_fstat_r
: A reentrant version of fstat. It takes a pointer to the global data block, which holds errno.

```
int _fstat_r(void *reent,
```

```
int fd, struct stat *pstat);
```

`_link_r` A reentrant version of `link`. It takes a pointer to the global data block, which holds `errno`.

```
int _link_r(void *reent,
    const char *old, const char *new);
```

`_unlink_r`
A reentrant version of `unlink`. It takes a pointer to the global data block, which holds `errno`.

```
int _unlink_r(void *reent, const char *file);
```

`_sbrk_r` A reentrant version of `sbrk`. It takes a pointer to the global data block, which holds `errno`.

```
char *_sbrk_r(void *reent, size_t incr);
```

10 Variable Argument Lists

The `printf` family of functions is defined to accept a variable number of arguments, rather than a fixed argument list. You can define your own functions with a variable argument list, by using macro definitions from either 'stdarg.h' (for compatibility with ANSI C) or from 'varargs.h' (for compatibility with a popular convention prior to ANSI C).

10.1 ANSI-standard macros, 'stdarg.h'

In ANSI C, a function has a variable number of arguments when its parameter list ends in an ellipsis (...). The parameter list must also include at least one explicitly named argument; that argument is used to initialize the variable list data structure.

ANSI C defines three macros (`va_start`, `va_arg`, and `va_end`) to operate on variable argument lists. 'stdarg.h' also defines a special type to represent variable argument lists: this type is called `va_list`.

10.1.1 Initialize variable argument list

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, rightmost);
```

Description

Use `va_start` to initialize the variable argument list *ap*, so that `va_arg` can extract values from it. *rightmost* is the name of the last explicit argument in the parameter list (the argument immediately preceding the ellipsis '...' that flags variable arguments in an ANSI C function header). You can only use `va_start` in a function declared using this ellipsis notation (not, for example, in one of its subfunctions).

Returns

`va_start` does not return a result.

Portability

ANSI C requires `va_start`.

10.1.2 Extract a value from argument list

Synopsis

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list *ap* (which you must previously create with *va_start*). Specify the type for the value as the second parameter to the macro, *type*.

You may pass a `va_list` object *ap* to a subfunction, and use `va_arg` from the subfunction rather than from the function actually declared with an ellipsis in the header; however, in that case you may *only* use `va_arg` from the subfunction. ANSI C does not permit extracting successive values from a single variable-argument list from different levels of the calling stack.

There is no mechanism for testing whether there is actually a next argument available; you might instead pass an argument count (or some other data that implies an argument count) as one of the fixed arguments in your function call.

Returns

`va_arg` returns the next argument, an object of type *type*.

Portability

ANSI C requires `va_arg`.

10.1.3 Abandon a variable argument list

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

Use `va_end` to declare that your program will not use the variable argument list *ap* any further.

Returns

`va_end` does not return a result.

Portability

ANSI C requires `va_end`.

10.2 Traditional macros, '`varargs.h`'

If your C compiler predates ANSI C, you may still be able to use variable argument lists using the macros from the '`varargs.h`' header file. These macros resemble their ANSI counterparts, but have important differences in usage. In particular, since traditional C has no declaration mechanism for variable argument lists, two additional macros are provided simply for the purpose of defining functions with variable argument lists.

As with '`stdarg.h`', the type `va_list` is used to hold a data structure representing a variable argument list.

10.2.1 Declare variable arguments

Synopsis

```
#include <varargs.h>
function(va_alist)
va_dcl
```

Description

To use the 'varargs.h' version of variable argument lists, you must declare your function with a call to the macro `va_alist` as its argument list, and use `va_dcl` as the declaration. *Do not use a semicolon after* `va_dcl`.

Returns

These macros cannot be used in a context where a return is syntactically possible.

Portability

va_alist and *va_dcl* were the most widespread method of declaring variable argument lists prior to ANSI C.

10.2.2 Initialize variable argument list

Synopsis

```
#include <varargs.h>
va_list ap;
va_start(ap);
```

Description

With the ‘`varargs.h`’ macros, use `va_start` to initialize a data structure *ap* to permit manipulating a variable argument list. *ap* must have the type *va_alist*.

Returns

`va_start` does not return a result.

Portability

`va_start` is also defined as a macro in ANSI C, but the definitions are incompatible; the ANSI version has another parameter besides *ap*.

10.2.3 Extract a value from argument list

Synopsis

```
#include <varargs.h>
type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list *ap* (which you must previously create with *va_start*). Specify the type for the value as the second parameter to the macro, *type*.

Returns

`va_arg` returns the next argument, an object of type *type*.

Portability

The `va_arg` defined in '`varargs.h`' has the same syntax and usage as the ANSI C version from '`stdarg.h`'.

10.2.4 Abandon a variable argument list

Synopsis

```
#include <varargs.h>
va_end(va_list ap);
```

Description

Use `va_end` to declare that your program will not use the variable argument list *ap* any further.

Returns

`va_end` does not return a result.

Portability

The `va_end` defined in ‘`varargs.h`’ has the same syntax and usage as the ANSI C version from ‘`stdarg.h`’.

Index

_

_asctime_r ... 132
_calloc_r ... 10
_close_r ... 158
_exit ... 153
_fdopen_r ... 59
_fopen_r ... 57
_fork_r ... 158
_free_r ... 19
_fstat_r ... 158
_getchar_r ... 69
_gets_r ... 70
_impure_ptr ... 149
_link_r ... 159
_localeconv_r ... 146
_lseek_r ... 158
_malloc_r ... 19
_mkstemp_r ... 72
_mktemp_r ... 72
_open_r ... 158
_perror_r ... 73
_putchar_r ... 75
_puts_r ... 76
_raise_r ... 128, 129
_rand_r ... 23
_read_r ... 158
_realloc_r ... 19
_reent ... 149
_rename_r ... 78
_sbrk_r ... 159
_setlocale_r ... 146
_signal_r ... 129
_srand_r ... 23
_stat_r ... 158
_strtod_r ... 24
_strtok_r ... 125
_strtol_r ... 25
_strtoul_r ... 27
_system_r ... 29
_tempnam_r ... 94
_tmpfile_r ... 93
_tmpnam_r ... 94
_tolower ... 44
_toupper ... 45
_unlink_r ... 159
_wait_r ... 158
_write_r ... 158

A

abort ... 2
abs ... 3
asctime ... 132
assert ... 4
atexit ... 5
atof ... 6
atoff ... 6
atoi ... 7
atol ... 7

B

bcmp ... 98
bsearch ... 9
bzero ... 100

C

calloc ... 10
clearerr ... 48
clock ... 133
close ... 153
ctime ... 134

D

difftime ... 135
div ... 11

E

ecvt ... 12
ecvtbuf ... 14
environ ... 16, 154
errno global vs macro ... 153
execve ... 154
exit ... 15
extra argument, reentrant fns ... 149

F

fclose 49
fcvt 12
fcvtbuf 14
fdopen 59
feof 50
ferror 51
fflush 52
fgetc 53
fgetpos 54
fgets 55
fiprintf 56
fopen 57
fork 154
fprintf 84
fputc 60
fputs 61
fread 62
free 19
freopen 63
fscanf 88
fseek 64
fsetpos 65
fstat 154
ftell 66
fwrite 67

G

gcvt 13
gcvtf 13
getc 68
getchar 69
getenv 16
getpid 154
gets 70
global reentrancy structure 149
gmtime 136

I

index 101
iprintf 71
isalnum 32
isalpha 33
isascii 34
isatty 154
iscntrl 35
isdigit 36
isgraph 38
islower 37
isprint 38
ispunct 39
isspace 40
isupper 41
isxdigit 42

K

kill 155

L

labs 17
ldiv 18
link 155
linking the C library 153
list of reentrant functions 149
localeconv 146
localtime 137
lseek 155

M

malloc 19
mbtowc 21
memchr 102
memcmp 103
memmove 105
memset 106
mkstemp 72
mktemp 72
mktime 138

O

OS interface subroutines 153

P

perror 73
printf 84
putc 74
putchar 75
puts 76

Q

qsort 22

R

raise 128, 129
rand 23
read 155
realloc 19
reent.h 149
reentrancy 149
reentrancy structure 149
reentrant function list 149
remove 77
rename 78
rewind 79
rindex 107

S

sbrk 155
scanf 88
setbuf 80
setlocale 146
setvbuf 81
signal 129
siprintf 83
sprintf 84
srand 23
sscanf 88
stat 156
strcat 108
strchr 109
strcmp 110
strcoll 111
strcpy 112
strcspn 113
strerror 114
strftime 139
strlen 117
strncat 118
strncmp 119
strncpy 120
strpbrk 121
strrchr 122
strspn 123
strstr 124
strtod 24
strtodf 24
strtok 125
strtol 25
strtoul 27
strxfrm 126
stubs 153
subroutines for OS interface 153
system 29

T

tempnam 94
time 141
times 156
tmpfile 93
tmpnam 94
toascii 43
tolower 44
toupper 45

U

unlink 156

V

va_alist 165
va_arg 163, 167
va_dcl 165
va_end 164, 168
va_start 162, 166
vfprintf 96
vprintf 96
vsprintf 96

W

wait 156
wctomb 30
write 156

The body of this manual is set in
pncr at 10.95pt,
with headings in **pncb at 10.95pt**
and examples in `pcrr`.
pncri at 10.95pt and
`pcrro`
are used for emphasis.

The Cygnus C Math Library

`libm` 1.4
May 1993

Steve Chamberlain
Roland Pesch
Cygnus Support

Cygnus Support
sac@cygnus.com
pesch@cygnus.com

Copyright © 1992, 1993 Cygnus Support

'libm' includes software developed by the University of California, Berkeley and its contributors.

'libm' includes software developed by Martin Jackson, Graham Haley and Steve Chamberlain of Tadpole Technology and released to Cygnus.

'libm' includes software developed at SunPro, a Sun Microsystems, Inc. business. Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, subject to the terms of the GNU General Public License, which includes the provision 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 manual into another language, under the above conditions for modified versions.

Table of Contents

1 Mathematical Functions (‘math.h’) **1**

1.1 Version of library 2
1.2 acos, acosf—arc cosine 3
1.3 acosh, acoshf—inverse hyperbolic cosine 4
1.4 asin, asinf—arc sine 5
1.5 asinh, asinhf—inverse hyperbolic sine 6
1.6 atan, atanf—arc tangent................................ 7
1.7 atan2, atan2f—arc tangent of y/x 8
1.8 atanh, atanhf—inverse hyperbolic tangent 9
1.9 jN,jNf,yN,yNf—Bessel functions 10
1.10 cbrt, cbrtf—cube root 11
1.11 copysign, copysignf—sign of *y*, magnitude of *x* 12
1.12 cosh, coshf—hyperbolic cosine 13
1.13 erf, erff, erfc, erfcf—error function 14
1.14 exp, expf—exponential 15
1.15 expm1, expm1f—exponential minus 1 16
1.16 fabs, fabsf—absolute value (magnitude).............. 17
1.17 floor, floorf, ceil, ceilf—floor and ceiling 18
1.18 fmod, fmodf—floating-point remainder (modulo)....... 19
1.19 frexp, frexpf—split floating-point number 20
1.20 gamma, gammaf, lgamma, lgammaf, gamma_r, ... 21
1.21 hypot, hypotf—distance from origin................... 23
1.22 ilogb, ilogbf—get exponent of floating point number 24
1.23 infinity, infinityf—representation of infinity 25
1.24 isnan,isnanf,isinf,isinff,finite,finitef—test for exceptional numbers 26
1.25 ldexp, ldexpf—load exponent 27
1.26 log, logf—natural logarithms......................... 28
1.27 log10, log10f—base 10 logarithms.................... 29
1.28 log1p, log1pf—log of 1 + *x* 30
1.29 matherr—modifiable math error handler 31
1.30 modf, modff—split fractional and integer parts 33
1.31 nan, nanf—representation of infinity 34
1.32 nextafter, nextafterf—get next number 35
1.33 pow, powf—x to the power y 36
1.34 rint, rintf, remainder, remainderf—round and remainder 37
1.35 scalbn, scalbnf—scale by integer..................... 38
1.36 sqrt, sqrtf—positive square root 39
1.37 sin, sinf, cos, cosf—sine or cosine 40

1.38 `sinh`, `sinhf`—hyperbolic sine 41
1.39 `tan`, `tanf`—tangent 42
1.40 `tanh`, `tanhf`—hyperbolic tangent 43

2 Reentrancy Properties of `libm` 45

Index 47

1 Mathematical Functions (‘math.h’)

This chapter groups a wide variety of mathematical functions. The corresponding definitions and declarations are in ‘math.h’. Two definitions from ‘math.h’ are of particular interest.

1. The representation of infinity as a `double` is defined as `HUGE_VAL`; this number is returned on overflow by many functions.
2. The structure `exception` is used when you write customized error handlers for the mathematical functions. You can customize error handling for most of these functions by defining your own version of `matherr`; see the section on `matherr` for details.

Since the error handling code calls `fputs`, the mathematical subroutines require stubs or minimal implementations for the same list of OS subroutines as `fputs`: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. See section “System Calls” in *The Cygnus C Support Library*, for a discussion and for sample minimal implementations of these support subroutines.

Alternative declarations of the mathematical functions, which exploit specific machine capabilities to operate faster—but generally have less error checking and may reflect additional limitations on some machines—are available when you include ‘fastmath.h’ instead of ‘math.h’.

1.1 Version of library

There are four different versions of the math library routines: IEEE, POSIX, X/Open, or SVID. The version may be selected at runtime by setting the global variable `_LIB_VERSION`, defined in 'math.h'. It may be set to one of the following constants defined in 'math.h': `_IEEE_`, `_POSIX_`, `_XOPEN_`, or `_SVID_`. The `_LIB_VERSION` variable is not specific to any thread, and changing it will affect all threads.

The versions of the library differ only in how errors are handled.

In IEEE mode, the `matherr` function is never called, no warning messages are printed, and `errno` is never set.

In POSIX mode, `errno` is set correctly, but the `matherr` function is never called and no warning messages are printed.

In X/Open mode, `errno` is set correctly, and `matherr` is called, but warning message are not printed.

In SVID mode, functions which overflow return 3.40282346638528860e+38, the maximum single precision floating point value, rather than infinity. Also, `errno` is set correctly, `matherr` is called, and, if `matherr` returns 0, warning messages are printed for some errors. For example, by default '`log(-1.0)`' writes this message on standard error output:

```
log: DOMAIN error
```

The library is set to X/Open mode by default.

1.2 acos, acosf—arc cosine

Synopsis

```
#include <math.h>
double acos(double x);
float acosf(float x);
```

Description

acos computes the inverse cosine (arc cosine) of the input value. Arguments to acos must be in the range -1 to 1.

acosf is identical to acos, except that it performs its calculations on floats.

Returns

acos and acosf return values in radians, in the range of 0 to π.

If x is not between -1 and 1, the returned value is NaN (not a number) the global variable errno is set to EDOM, and a DOMAIN error message is sent as standard error output.

You can modify error handling for these functions using matherr.

1.3 acosh, acoshf—inverse hyperbolic cosine

Synopsis

```
#include <math.h>
double acosh(double x);
float acoshf(float x);
```

Description

`acosh` calculates the inverse hyperbolic cosine of x. `acosh` is defined as

$$ln\left(x + \sqrt{x^2 - 1}\right)$$

x must be a number greater than or equal to 1.

`acoshf` is identical, other than taking and returning floats.

Returns

`acosh` and `acoshf` return the calculated value. If x less than 1, the return value is NaN and `errno` is set to `EDOM`.

You can change the error-handling behavior with the non-ANSI `matherr` function.

Portability

Neither `acosh` nor `acoshf` are ANSI C. They are not recommended for portable programs.

1.4 asin, asinf—arc sine

Synopsis

```
#include <math.h>
double asin(double x);
float asinf(float x);
```

Description

asin computes the inverse sine (arc sine) of the argument x. Arguments to asin must be in the range -1 to 1.

asinf is identical to asin, other than taking and returning floats.

You can modify error handling for these routines using matherr.

Returns

asin returns values in radians, in the range of $-\pi/2$ to $\pi/2$.

If x is not in the range -1 to 1, asin and asinf return NaN (not a number), set the global variable errno to EDOM, and issue a DOMAIN error message.

You can change this error treatment using matherr.

1.5 `asinh`, `asinhf`—inverse hyperbolic sine

Synopsis

```
#include <math.h>
double asinh(double x);
float asinhf(float x);
```

Description

`asinh` calculates the inverse hyperbolic sine of x. `asinh` is defined as

$$sign(x) \times ln\left(|x| + \sqrt{1 + x^2}\right)$$

`asinhf` is identical, other than taking and returning floats.

Returns

`asinh` and `asinhf` return the calculated value.

Portability

Neither `asinh` nor `asinhf` are ANSI C.

1.6 atan, atanf—arc tangent

Synopsis

```
#include <math.h>
double atan(double x);
float atanf(float x);
```

Description

`atan` computes the inverse tangent (arc tangent) of the input value.
`atanf` is identical to `atan`, save that it operates on `floats`.

Returns

`atan` returns a value in radians, in the range of $-\pi/2$ to $\pi/2$.

Portability

`atan` is ANSI C. `atanf` is an extension.

1.7 `atan2`, `atan2f`—arc tangent of y/x

Synopsis

```
#include <math.h>
double atan2(double y,double x);
float atan2f(float y,float x);
```

Description

`atan2` computes the inverse tangent (arc tangent) of y/x. `atan2` produces the correct result even for angles near $\pi/2$ or $-\pi/2$ (that is, when x is near 0).

`atan2f` is identical to `atan2`, save that it takes and returns `float`.

Returns

`atan2` and `atan2f` return a value in radians, in the range of $-\pi$ to π.

If both x and y are 0.0, `atan2` causes a `DOMAIN` error.

You can modify error handling for these functions using `matherr`.

Portability

`atan2` is ANSI C. `atan2f` is an extension.

1.8 `atanh`, `atanhf`—inverse hyperbolic tangent

Synopsis

```
#include <math.h>
double atanh(double x);
float atanhf(float x);
```

Description

`atanh` calculates the inverse hyperbolic tangent of *x*.

`atanhf` is identical, other than taking and returning `float` values.

Returns

`atanh` and `atanhf` return the calculated value.

If $|x|$ is greater than 1, the global `errno` is set to `EDOM` and the result is a NaN. A `DOMAIN error` is reported.

If $|x|$ is 1, the global `errno` is set to `EDOM`; and the result is infinity with the same sign as `x`. A `SING error` is reported.

You can modify the error handling for these routines using `matherr`.

Portability

Neither `atanh` nor `atanhf` are ANSI C.

1.9 `jN`,`jNf`,`yN`,`yNf`—Bessel functions

Synopsis

```
#include <math.h>
double j0(double x);
float j0f(float x);
double j1(double x);
float j1f(float x);
double jn(int n, double x);
float jnf(int n, float x);
double y0(double x);
float y0f(float x);
double y1(double x);
float y1f(float x);
double yn(int n, double x);
float ynf(int n, float x);
```

Description

The Bessel functions are a family of functions that solve the differential equation

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 - p^2) y = 0$$

These functions have many applications in engineering and physics.

`jn` calculates the Bessel function of the first kind of order n. `j0` and `j1` are special cases for order 0 and order 1 respectively.

Similarly, `yn` calculates the Bessel function of the second kind of order n, and `y0` and `y1` are special cases for order 0 and 1.

`jnf`, `j0f`, `j1f`, `ynf`, `y0f`, and `y1f` perform the same calculations, but on `float` rather than `double` values.

Returns

The value of each Bessel function at x is returned.

Portability

None of the Bessel functions are in ANSI C.

1.10 `cbrt`, `cbrtf`—cube root

Synopsis

```
#include <math.h>
double cbrt(double x);
float  cbrtf(float x);
```

Description

`cbrt` computes the cube root of the argument.

Returns

The cube root is returned.

Portability

`cbrt` is in System V release 4. `cbrtf` is an extension.

1.11 `copysign`, `copysignf`—sign of *y*, magnitude of *x*

Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

Description

`copysign` constructs a number with the magnitude (absolute value) of its first argument, *x*, and the sign of its second argument, *y*.

`copysignf` does the same thing; the two functions differ only in the type of their arguments and result.

Returns

`copysign` returns a `double` with the magnitude of *x* and the sign of *y*. `copysignf` returns a `float` with the magnitude of *x* and the sign of *y*.

Portability

`copysign` is not required by either ANSI C or the System V Interface Definition (Issue 2).

1.12 cosh, coshf—hyperbolic cosine

Synopsis

```
#include <math.h>
double cosh(double x);
float coshf(float x)
```

Description

`cosh` computes the hyperbolic cosine of the argument *x*. `cosh(x)` is defined as

$$\frac{(e^x + e^{-x})}{2}$$

Angles are specified in radians. `coshf` is identical, save that it takes and returns `float`.

Returns

The computed value is returned. When the correct value would create an overflow, `cosh` returns the value `HUGE_VAL` with the appropriate sign, and the global value `errno` is set to `ERANGE`.

You can modify error handling for these functions using the function `matherr`.

Portability

`cosh` is ANSI. `coshf` is an extension.

1.13 `erf`, `erff`, `erfc`, `erfcf`—error function

Synopsis

```
#include <math.h>
double erf(double x);
float erff(float x);
double erfc(double x);
float erfcf(float x);
```

Description

`erf` calculates an approximation to the “error function”, which estimates the probability that an observation will fall within x standard deviations of the mean (assuming a normal distribution). The error function is defined as

$$\frac{2}{\sqrt{\pi}} \times \int_0^x e^{-t^2} dt$$

`erfc` calculates the complementary probability; that is, `erfc(x)` is `1 - erf(x)`. `erfc` is computed directly, so that you can use it to avoid the loss of precision that would result from subtracting large probabilities (on large x) from 1.

`erff` and `erfcf` differ from `erf` and `erfc` only in the argument and result types.

Returns

For positive arguments, `erf` and all its variants return a probability—a number between 0 and 1.

Portability

None of the variants of `erf` are ANSI C.

1.14 exp, expf—exponential

Synopsis

```
#include <math.h>
double exp(double x);
float expf(float x);
```

Description

exp and expf calculate the exponential of x, that is, e^x (where e is the base of the natural system of logarithms, approximately 2.71828).

You can use the (non-ANSI) function matherr to specify error handling for these functions.

Returns

On success, exp and expf return the calculated value. If the result underflows, the returned value is 0. If the result overflows, the returned value is HUGE_VAL. In either case, errno is set to ERANGE.

Portability

exp is ANSI C. expf is an extension.

1.15 `expm1`, `expm1f`—exponential minus 1

Synopsis

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
```

Description

`expm1` and `expm1f` calculate the exponential of x and subtract 1, that is, $e^x - 1$ (where e is the base of the natural system of logarithms, approximately 2.71828). The result is accurate even for small values of x, where using `exp(x)-1` would lose many significant digits.

Returns

e raised to the power x, minus 1.

Portability

Neither `expm1` nor `expm1f` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.16 fabs, fabsf—absolute value (magnitude)

Synopsis

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
```

Description

fabs and fabsf calculate $|x|$, the absolute value (magnitude) of the argument *x*, by direct manipulation of the bit representation of *x*.

Returns

The calculated value is returned. No errors are detected.

Portability

fabs is ANSI. fabsf is an extension.

1.17 `floor`, `floorf`, `ceil`, `ceilf`—floor and ceiling

Synopsis

```
#include <math.h>
double floor(double x);
float floorf(float x);
double ceil(double x);
float ceilf(float x);
```

Description

`floor` and `floorf` find $\lfloor x \rfloor$, the nearest integer less than or equal to x. `ceil` and `ceilf` find $\lceil x \rceil$, the nearest integer greater than or equal to x.

Returns

`floor` and `ceil` return the integer result as a double. `floorf` and `ceilf` return the integer result as a float.

Portability

`floor` and `ceil` are ANSI. `floorf` and `ceilf` are extensions.

1.18 `fmod`, `fmodf`—floating-point remainder (modulo)

Synopsis

```
#include <math.h>
double fmod(double x, double y)
float fmodf(float x, float y)
```

Description

The `fmod` and `fmodf` functions compute the floating-point remainder of *x*/*y* (*x* modulo *y*).

Returns

The `fmod` function returns the value $x - i \times y$, for the largest integer *i* such that, if *y* is nonzero, the result has the same sign as *x* and magnitude less than the magnitude of *y*.

`fmod(x,0)` returns NaN, and sets `errno` to `EDOM`.

You can modify error treatment for these functions using `matherr`.

Portability

`fmod` is ANSI C. `fmodf` is an extension.

1.19 `frexp`, `frexpf`—split floating-point number

Synopsis

```
#include <math.h>
double frexp(double val, int *exp);
float frexpf(float val, int *exp);
```

Description

All non zero, normal numbers can be described as *m* * 2***p*. `frexp` represents the double *val* as a mantissa *m* and a power of two *p*. The resulting mantissa will always be greater than or equal to `0.5`, and less than `1.0` (as long as *val* is nonzero). The power of two will be stored in `*`*exp*.

m and *p* are calculated so that $val = m \times 2^p$.

`frexpf` is identical, other than taking and returning floats rather than doubles.

Returns

`frexp` returns the mantissa *m*. If *val* is `0`, infinity, or Nan, `frexp` will set `*`*exp* to `0` and return *val*.

Portability

`frexp` is ANSI. `frexpf` is an extension.

1.20 gamma, gammaf, lgamma, lgammaf, gamma_r,

Synopsis

```
#include <math.h>
double gamma(double x);
float gammaf(float x);
double lgamma(double x);
float lgammaf(float x);
double gamma_r(double x, int *signgamp);
float gammaf_r(float x, int *signgamp);
double lgamma_r(double x, int *signgamp);
float lgammaf_r(float x, int *signgamp);
```

Description

`gamma` calculates $ln(\Gamma(x))$, the natural logarithm of the gamma function of *x*. The gamma function (`exp(gamma(x))`) is a generalization of factorial, and retains the property that $\Gamma(N) \equiv N \times \Gamma(N-1)$. Accordingly, the results of the gamma function itself grow very quickly. `gamma` is defined as $ln(\Gamma(x))$ rather than simply $\Gamma(x)$ to extend the useful range of results representable.

The sign of the result is returned in the global variable `signgam`, which is declared in math.h.

`gammaf` performs the same calculation as `gamma`, but uses and returns `float` values.

`lgamma` and `lgammaf` are alternate names for `gamma` and `gammaf`. The use of `lgamma` instead of `gamma` is a reminder that these functions compute the log of the gamma function, rather than the gamma function itself.

The functions `gamma_r`, `gammaf_r`, `lgamma_r`, and `lgammaf_r` are just like `gamma`, `gammaf`, `lgamma`, and `lgammaf`, respectively, but take an additional argument. This additional argument is a pointer to an integer. This additional argument is used to return the sign of the result, and the global variable `signgam` is not used. These functions may be used for reentrant calls (but they will still set the global variable `errno` if an error occurs).

Do not confuse the function `gamma_r`, which takes an additional argument which is a pointer to an integer, with the function `_gamma_r`, which takes an additional argument which is a pointer to a reentrancy structure.

Returns

Normally, the computed result is returned.

When *x* is a nonpositive integer, `gamma` returns `HUGE_VAL` and `errno` is set to `EDOM`. If the result overflows, `gamma` returns `HUGE_VAL` and `errno` is set to `ERANGE`.

You can modify this error treatment using `matherr`.

Portability

Neither `gamma` nor `gammaf` is ANSI C.

1.21 hypot, hypotf—distance from origin

Synopsis

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
```

Description

`hypot` calculates the Euclidean distance $\sqrt{x^2 + y^2}$ between the origin (0,0) and a point represented by the Cartesian coordinates (*x*,*y*). `hypotf` differs only in the type of its arguments and result.

Returns

Normally, the distance value is returned. On overflow, `hypot` returns `HUGE_VAL` and sets `errno` to `ERANGE`.

You can change the error treatment with `matherr`.

Portability

`hypot` and `hypotf` are not ANSI C.

1.22 `ilogb`, `ilogbf`—get exponent of floating point number

Synopsis

```
#include <math.h>
int ilogb(double val);
int ilogbf(float val);
```

Description

All non zero, normal numbers can be described as *m* * 2***p*. `ilogb` and `ilogbf` examine the argument *val*, and return *p*. The functions `frexp` and `frexpf` are similar to `ilogb` and `ilogbf`, but also return *m*.

Returns

`ilogb` and `ilogbf` return the power of two used to form the floating point argument. If *val* is `0`, they return `- INT_MAX` (`INT_MAX` is defined in limits.h). If *val* is infinite, or NaN, they return `INT_MAX`.

Portability

Neither `ilogb` nor `ilogbf` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.23 `infinity`, `infinityf`—representation of infinity

Synopsis

```
#include <math.h>
double infinity(void);
float infinityf(void);
```

Description

`infinity` and `infinityf` return the special number IEEE infinity in double and single precision arithmetic respectivly.

1.24 `isnan`,`isnanf`,`isinf`,`isinff`,`finite`,`finitef`—test for exceptional numbers

Synopsis

```
#include <ieeefp.h>
int isnan(double arg);
int isinf(double arg);
int finite(double arg);
int isnanf(float arg);
int isinff(float arg);
int finitef(float arg);
```

Description

These functions provide information on the floating point argument supplied.

There are five major number formats -

`zero` a number which contains all zero bits.

`subnormal`
Is used to represent number with a zero exponent, but a non zero fraction.

`normal` A number with an exponent, and a fraction

`infinity` A number with an all 1's exponent and a zero fraction.

`NAN` A number with an all 1's exponent and a non zero fraction.

`isnan` returns 1 if the argument is a nan. `isinf` returns 1 if the argument is infinity. `finite` returns 1 if the argument is zero, subnormal or normal. The `isnanf`, `isinff` and `finitef` perform the same operations as their `isnan`, `isinf` and `finite` counterparts, but on single precision floating point numbers.

1.25 `ldexp`, `ldexpf`—load exponent

Synopsis

```
#include <math.h>
double ldexp(double val, int exp);
float ldexpf(float val, int exp);
```

Description

`ldexp` calculates the value $val \times 2^{exp}$. `ldexpf` is identical, save that it takes and returns `float` rather than `double` values.

Returns

`ldexp` returns the calculated value.

Underflow and overflow both set `errno` to `ERANGE`. On underflow, `ldexp` and `ldexpf` return 0.0. On overflow, `ldexp` returns plus or minus `HUGE_VAL`.

Portability

`ldexp` is ANSI, `ldexpf` is an extension.

1.26 `log`, `logf`—natural logarithms

Synopsis

```
#include <math.h>
double log(double x);
float logf(float x);
```

Description

Return the natural logarithm of *x*, that is, its logarithm base e (where e is the base of the natural system of logarithms, 2.71828...). `log` and `logf` are identical save for the return and argument types.

You can use the (non-ANSI) function `matherr` to specify error handling for these functions.

Returns

Normally, returns the calculated value. When *x* is zero, the returned value is `-HUGE_VAL` and `errno` is set to `ERANGE`. When *x* is negative, the returned value is `-HUGE_VAL` and `errno` is set to `EDOM`. You can control the error behavior via `matherr`.

Portability

`log` is ANSI, `logf` is an extension.

1.27 `log10`, `log10f`—base 10 logarithms

Synopsis

```
#include <math.h>
double log10(double x);
float log10f(float x);
```

Description

`log10` returns the base 10 logarithm of *x*. It is implemented as `log(x) / log(10)`.

`log10f` is identical, save that it takes and returns `float` values.

Returns

`log10` and `log10f` return the calculated value.

See the description of `log` for information on errors.

Portability

`log10` is ANSI C. `log10f` is an extension.

1.28 `log1p`, `log1pf`—log of `1 + x`

Synopsis

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
```

Description

`log1p` calculates $ln(1+x)$, the natural logarithm of `1+x`. You can use `log1p` rather than '`log(1+x)`' for greater precision when *x* is very small.

`log1pf` calculates the same thing, but accepts and returns `float` values rather than `double`.

Returns

`log1p` returns a `double`, the natural log of `1+x`. `log1pf` returns a `float`, the natural log of `1+x`.

Portability

Neither `log1p` nor `log1pf` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.29 `matherr`—modifiable math error handler

Synopsis

```
#include <math.h>
int matherr(struct exception *e);
```

Description

`matherr` is called whenever a math library function generates an error. You can replace `matherr` by your own subroutine to customize error treatment. The customized `matherr` must return 0 if it fails to resolve the error, and non-zero if the error is resolved.

When `matherr` returns a nonzero value, no error message is printed and the value of `errno` is not modified. You can accomplish either or both of these things in your own `matherr` using the information passed in the structure `*e`.

This is the `exception` structure (defined in ‘`math.h`’):

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
int err;
};
```

The members of the exception structure have the following meanings:

`type` The type of mathematical error that occured; macros encoding error types are also defined in ‘`math.h`’.

`name` a pointer to a null-terminated string holding the name of the math library function where the error occurred.

`arg1, arg2`
The arguments which caused the error.

`retval` The error return value (what the calling function will return).

`err` If set to be non-zero, this is the new value assigned to `errno`.

The error types defined in ‘`math.h`’ represent possible mathematical errors as follows:

`DOMAIN` An argument was not in the domain of the function; e.g. `log(-1.0)`.

`SING` The requested calculation would result in a singularity; e.g. `pow(0.0,-2.0)`

`OVERFLOW` A calculation would produce a result too large to represent; e.g. `exp(1000.0)`.

`UNDERFLOW`
: A calculation would produce a result too small to represent; e.g. `exp(-1000.0)`.

`TLOSS`
: Total loss of precision. The result would have no significant digits; e.g. `sin(10e70)`.

`PLOSS`
: Partial loss of precision.

Returns

The library definition for `matherr` returns `0` in all cases.

You can change the calling function's result from a customized `matherr` by modifying `e->retval`, which propagates backs to the caller.

If `matherr` returns `0` (indicating that it was not able to resolve the error) the caller sets `errno` to an appropriate value, and prints an error message.

Portability

`matherr` is not ANSI C.

1.30 `modf`, `modff`—split fractional and integer parts

Synopsis

```
#include <math.h>
double modf(double val, double *ipart);
float modff(float val, float *ipart);
```

Description

`modf` splits the double *val* apart into an integer part and a fractional part, returning the fractional part and storing the integer part in `*`*ipart*. No rounding whatsoever is done; the sum of the integer and fractional parts is guaranteed to be exactly equal to *val*. That is, if . *realpart* = modf(*val*, &*intpart*); then ‘*realpart*+*intpart*’ is the same as *val*. `modff` is identical, save that it takes and returns `float` rather than `double` values.

Returns

The fractional part is returned. Each result has the same sign as the supplied argument *val*.

Portability

`modf` is ANSI C. `modff` is an extension.

1.31 `nan`, `nanf`—representation of infinity

Synopsis

```
#include <math.h>
double nan(void);
float nanf(void);
```

Description

`nan` and `nanf` return an IEEE NaN (Not a Number) in double and single precision arithmetic respectivly.

1.32 `nextafter`, `nextafterf`—get next number

Synopsis

```
#include <math.h>
double nextafter(double val, double dir);
float nextafterf(float val, float dir);
```

Description

`nextafter` returns the double) precision floating point number closest to *val* in the direction toward *dir*. `nextafterf` performs the same operation in single precision. For example, `nextafter(0.0,1.0)` returns the smallest positive number which is representable in double precision.

Returns

Returns the next closest number to *val* in the direction toward *dir*.

Portability

Neither `nextafter` nor `nextafterf` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.33 `pow`, `powf`—x to the power y

Synopsis

```
#include <math.h>
double pow(double x, double y);
float pow(float x, float y);
```

Description

`pow` and `powf` calculate *x* raised to the exp1.0nt *y*. (That is, x^y.)

Returns

On success, `pow` and `powf` return the value calculated.

When the argument values would produce overflow, `pow` returns `HUGE_VAL` and set `errno` to `ERANGE`. If the argument *x* passed to `pow` or `powf` is a negative noninteger, and *y* is also not an integer, then `errno` is set to `EDOM`. If *x* and *y* are both 0, then `pow` and `powf` return `1`.

You can modify error handling for these functions using `matherr`.

Portability

`pow` is ANSI C. `powf` is an extension.

1.34 `rint`, `rintf`, `remainder`, `remainderf`—round and remainder

Synopsis

```
#include <math.h>
double rint(double x);
float rintf(float x);
double remainder(double x, double y);
float remainderf(float x, float y);
```

Description

`rint` and `rintf` returns their argument rounded to the nearest integer. `remainder` and `remainderf` find the remainder of *x*/*y*; this value is in the range -*y*/2 .. +*y*/2.

Returns

`rint` and `remainder` return the integer result as a double.

Portability

`rint` and `remainder` are System V release 4. `rintf` and `remainderf` are extensions.

1.35 `scalbn`, `scalbnf`—scale by integer

Synopsis

```
#include <math.h>
double scalbn(double x, int y);
float scalbnf(float x, int y);
```

Description

`scalbn` and `scalbnf` scale *x* by *n*, returning *x* times 2 to the power *n*. The result is computed by manipulating the exponent, rather than by actually performing an exponentiation or multiplication.

Returns

x times 2 to the power *n*.

Portability

Neither `scalbn` nor `scalbnf` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.36 sqrt, sqrtf—positive square root

Synopsis

```
#include <math.h>
double sqrt(double x);
float  sqrtf(float x);
```

Description

sqrt computes the positive square root of the argument. You can modify error handling for this function with matherr.

Returns

On success, the square root is returned. If *x* is real and positive, then the result is positive. If *x* is real and negative, the global value errno is set to EDOM (domain error).

Portability

sqrt is ANSI C. sqrtf is an extension.

1.37 `sin`, `sinf`, `cos`, `cosf`—sine or cosine

Synopsis

```
#include <math.h>
double sin(double x);
float  sinf(float x);
double cos(double x);
float cosf(float x);
```

Description

`sin` and `cos` compute (respectively) the sine and cosine of the argument *x*. Angles are specified in radians.

`sinf` and `cosf` are identical, save that they take and return `float` values.

Returns

The sine or cosine of *x* is returned.

Portability

`sin` and `cos` are ANSI C. `sinf` and `cosf` are extensions.

1.38 `sinh`, `sinhf`—hyperbolic sine

Synopsis

```
#include <math.h>
double sinh(double x);
float  sinhf(float x);
```

Description

`sinh` computes the hyperbolic sine of the argument x. Angles are specified in radians. `sinh`(x) is defined as

$$\frac{e^x - e^{-x}}{2}$$

`sinhf` is identical, save that it takes and returns `float` values.

Returns

The hyperbolic sine of x is returned.

When the correct result is too large to be representable (an overflow), `sinh` returns `HUGE_VAL` with the appropriate sign, and sets the global value `errno` to `ERANGE`.

You can modify error handling for these functions with `matherr`.

Portability

`sinh` is ANSI C. `sinhf` is an extension.

1.39 `tan`, `tanf`—tangent

Synopsis

```
#include <math.h>
double tan(double x);
float tanf(float x);
```

Description

`tan` computes the tangent of the argument *x*. Angles are specified in radians.

`tanf` is identical, save that it takes and returns `float` values.

Returns

The tangent of *x* is returned.

Portability

`tan` is ANSI. `tanf` is an extension.

1.40 tanh, tanhf—hyperbolic tangent

Synopsis

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

Description

`tanh` computes the hyperbolic tangent of the argument *x*. Angles are specified in radians.

`tanh(x)` is defined as

```
sinh(x)/cosh(x)
```

`tanhf` is identical, save that it takes and returns `float` values.

Returns

The hyperbolic tangent of *x* is returned.

Portability

`tanh` is ANSI C. `tanhf` is an extension.

2 Reentrancy Properties of libm

When a math function detects an error, it sets the static variable errno. Depending upon the severity of the error, it may also print a message to stderr. None of this behavior is reentrant. When one process detects an error, it sets errno. If another process is testing errno, it detects the change and probably fails. Note that failing system calls can also set errno. This problem can only be fixed by either ignoring errno, or treating it as part of the context of a process and switching it along with the rest of a processor state. In normal debugged programs, there are usually no math subroutine errors—and therefore no matherr calls; in that situation, the math functions behave reentrantly.

As an alternative, you can use the reentrant versions of the mathematical functions: these versions have a different name, normally formed by prepending '_' and appending '_r', and use an extra argument—a pointer to the particular reentrancy structure to use. See section "Reentrancy" in *The Cygnus C Support Library*, for more discussion of this approach to reentrancy.

The reentrancy structure is always an additional first argument; for example, the reentrant version of 'double acos (double *x*)' is 'double _acos_r (void **reent*, double *x*)'.

Here is a list of the names for reentrant versions of the mathematical library functions:

```
_acos_r         _gammaf_r_r      _log10f_r
_acosf_r        _hypot_r         _log_r
_acosh_r        _hypotf_r        _logf_r
_acoshf_r       _j0_r            _pow_r
_asin_r         _j0f_r           _powf_r
_asinf_r        _j1_r            _remainder_r
_atanh_r        _j1f_r           _sinh_r
_atanhf_r       _jn_r            _sinhf_r
_cosh_r         _jnf_r           _sqrt_r
_coshf_r        _ldexp_r         _sqrtf_r
_exp_r          _ldexpf_r        _y0_r
_expf_r         _lgamma_r        _y0f_r
_fmod_r         _lgamma_r_r      _y1_r
_fmodf_r        _lgammaf_r       _y1f_r
_gamma_r        _lgammaf_r_r     _yn_r
_gamma_r_r      _log10_r         _ynf_r
_gammaf_r
```

Index

A

acos 3
acosf 3
acosh 4
acoshf 4
asin 5
asinf 5
asinh 6
asinhf 6
atan 7
atan2 8
atan2f 8
atanf 7
atanh 9
atanhf 9

C

cbrt 11
cbrtf 11
ceil 18
ceilf 18
copysign 12
copysignf 12
cos 40
cosf 40

E

erf 14
erfc 14
erfcf 14
erff 14
exp 15
expf 15
expm1 16
expm1f 16

F

fabs 17
fabsf 17
finite 26
finitef 26
floor 18
floorf 18
fmod 19
fmodf 19
frexp 20
frexpf 20

G

gamma 21
gamma_r 21
gammaf 21
gammaf_r 21

H

hypot 23
hypotf 23

I

ilogb 24
ilogbf 24
infinity 25
infinityf 25
isinf 26
isinff 26
isnan 26
isnanf 26

J

j0 10
j0f 10
j1 10
j1f 10
jn 10
jnf 10

L

ldexp 27
ldexpf 27
lgamma 21
lgamma_r 21
lgammaf 21
lgammaf_r 21

log 28
log10 29
log10f 29
log1p 30
log1pf 30
logf 28

M

matherr 31
matherr and reentrancy 45
modf 33
modff 33

N

nan 34
nanf 34
nextafter 35
nextafterf 35

O

OS stubs 1

P

pow 36
powf 36

R

reentrancy 45
remainder 37
remainderf 37
rint 37
rintf 37

S

scalbn 38
scalbnf 38
sin 40
sinf 40
sinh 41
sinhf 41
sqrt 39
sqrtf 39
stubs 1
support subroutines 1
system calls 1

T

tan 42
tanf 42
tanh 43
tanhf 43

Y

y0 10
y0f 10
y1 10
y1f 10
yn 10
ynf 10

The body of this manual is set in
pncr at 10.95pt,
with headings in **pncb at 10.95pt**
and examples in `pcrr`.
pncri at 10.95pt and
`pcrro`
are used for emphasis.

The C Preprocessor

Last revised July 1992
for GCC version 2

Richard M. Stallman

This booklet is eventually intended to form the first chapter of a GNU C Language manual.

Copyright © 1987, 1989, 1991, 1992, 1993, 1994, 1995 Free Software Foundation, Inc.

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

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

Table of Contents

1 The C Preprocessor 1
1.1 Transformations Made Globally 1
1.2 Preprocessing Directives 2
1.3 Header Files... 3
1.3.1 Uses of Header Files 3
1.3.2 The '#include' Directive 4
1.3.3 How '#include' Works 5
1.3.4 Once-Only Include Files......................... 6
1.3.5 Inheritance and Header Files 7
1.4 Macros... 8
1.4.1 Simple Macros 8
1.4.2 Macros with Arguments 10
1.4.3 Predefined Macros 12
1.4.3.1 Standard Predefined Macros.......... 12
1.4.3.2 Nonstandard Predefined Macros...... 15
1.4.4 Stringification................................. 17
1.4.5 Concatenation 18
1.4.6 Undefining Macros............................ 19
1.4.7 Redefining Macros 20
1.4.8 Pitfalls and Subtleties of Macros............... 20
1.4.8.1 Improperly Nested Constructs........ 21
1.4.8.2 Unintended Grouping of Arithmetic .. 21
1.4.8.3 Swallowing the Semicolon 22
1.4.8.4 Duplication of Side Effects............ 23
1.4.8.5 Self-Referential Macros............... 24
1.4.8.6 Separate Expansion of Macro Arguments
.... 25
1.4.8.7 Cascaded Use of Macros 27
1.4.9 Newlines in Macro Arguments................. 27
1.5 Conditionals.. 28
1.5.1 Why Conditionals are Used 28
1.5.2 Syntax of Conditionals......................... 29
1.5.2.1 The '#if' Directive.................... 29
1.5.2.2 The '#else' Directive 30
1.5.2.3 The '#elif' Directive 30
1.5.3 Keeping Deleted Code for Future Reference ... 31
1.5.4 Conditionals and Macros....................... 32
1.5.5 Assertions..................................... 33
1.5.6 The '#error' and '#warning' Directives......... 35
1.6 Combining Source Files 36
1.7 Miscellaneous Preprocessing Directives................. 37

1.8 C Preprocessor Output 37
1.9 Invoking the C Preprocessor 38

Concept Index 45

Index of Directives, Macros and Options 47

1 The C Preprocessor

The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in some details. This manual discusses the GNU C preprocessor, the C Compatible Compiler Preprocessor. The GNU C preprocessor provides a superset of the features of ANSI Standard C.

ANSI Standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. Strictly speaking, to get ANSI Standard C, you must use the options '`-trigraphs`', '`-undef`' and '`-pedantic`', but in practice the consequences of having strict ANSI Standard C make it undesirable to do this. See Section 1.9 "Invocation," page 38.

1.1 Transformations Made Globally

Most C preprocessor features are inactive unless you give specific directives to request their use. (Preprocessing directives are lines starting with '`#`'; see Section 1.2 "Directives," page 2). But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of directives.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.

- Predefined macro names are replaced with their expansions (see Section 1.4.3 “Predefined,” page 12).

The first two transformations are done *before* nearly all other parsing and before preprocessing directives are recognized. Thus, for example, you can split a line cosmetically with Backslash-Newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent into ‘#define FOO 1020’. You can split even an escape sequence with Backslash-Newline. For example, you can split "foo\bar" between the ‘\’ and the ‘b’ to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a Backslash can be inserted in a string constant as an ordinary character by writing a double Backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C does not allow Newlines in string constants, so they do not consider this a problem.)

But there are a few exceptions to all three transformations.

- C comments and predefined macro names are not recognized inside a ‘#include’ directive in which the file name is delimited with ‘<’ and ‘>’.
- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule, not an exception, but it is worth noting here anyway.)
- Backslash-Newline may not safely be used within an ANSI “trigraph”. Trigraphs are converted before Backslash-Newline is deleted. If you write what looks like a trigraph with a Backslash-Newline inside, the Backslash-Newline is deleted as usual, but it is then too late to recognize the trigraph.

  This exception is relevant only if you use the ‘-trigraphs’ option to enable trigraph processing. See Section 1.9 “Invocation,” page 38.

1.2 Preprocessing Directives

Most preprocessor features are active only if you use preprocessing directives to request their use.

Preprocessing directives are lines in your program that start with '#'. The '#' is followed by an identifier that is the *directive name*. For example, '#define' is the directive that defines a macro. Whitespace is also allowed before and after the '#'.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives.

Some directive names require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, '#define' must be followed by a macro name and the intended expansion of the macro. See Section 1.4.1 "Simple Macros," page 8.

A preprocessing directive cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the directive into multiple lines, but the comments are changed to Spaces before the directive is interpreted. The only way a significant Newline can occur in a preprocessing directive is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The '#' and the directive name cannot come from a macro expansion. For example, if 'foo' is defined as a macro expanding to 'define', that does not make '#foo' a valid preprocessing directive.

1.3 Header Files

A header file is a file containing C declarations and macro definitions (see Section 1.4 "Macros," page 8) to be shared between several source files. You request the use of a header file in your program with the C preprocessing directive '#include'.

1.3.1 Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with '.h'. Avoid unusual characters in header file names, as they reduce portability.

1.3.2 The '#include' Directive

Both user and system header files are included using the preprocessing directive '#include'. It has three variants:

#include <*file*>

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option '-I' (see Section 1.9 "Invocation," page 38). The option '-nostdinc' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of '#include' is slightly special because comments are not recognized within the '<...>'. Thus, in '#include <x/*y>' the '/*' does not start a comment and the directive specifies inclusion of a system header file named 'x/*y'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument *file* may not contain a '>' character. It may, however, contain a '<' character.

#include "*file*"

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the '-I-' option is used, the special treatment of the current directory is inhibited.)

The argument *file* may not contain '"' characters. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, '#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

#include *anything else*

This variant is called a *computed #include*. Any '#include' directive whose argument does not fit the above two forms is a computed include. The text *anything else* is checked for macro calls, which are expanded (see Section 1.4 "Macros," page 8). When this is done, the result must fit one of the above two variants—in particular, the expanded text must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-specific configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

1.3.3 How '#include' Works

The '#include' directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '#include' directive. For example, given a header file 'header.h' as follows,

```
char *test ();
```

and a main program called 'program.c' that uses the header file, like this,

```
int x;
#include "header.h"

main ()
{
  printf (test ());
}
```

the output generated by the C preprocessor for 'program.c' as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

It is possible for a header file to begin or end a syntactic unit such as a function definition, but that would be very confusing, so don't do it.

The line following the '`#include`' directive is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

1.3.4 Once-Only Include Files

Very often, one header file includes another. It can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, we often wish to prevent multiple inclusion of a header file.

The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

the entire file

#endif /* FILE_FOO_SEEN */
```

The macro `FILE_FOO_SEEN` indicates that the file has been included once already. In a user header file, the macro name should not begin with '_'. In a system header file, this name should begin with '__' to avoid conflicts with user programs. In any kind of header file, the macro

name should contain the name of the file and some additional text, to avoid conflicts with other header files.

The GNU C preprocessor is programmed to notice when a header file uses this particular construct and handle it efficiently. If a header file is contained entirely in a '#ifndef' conditional, then it records that fact. If a subsequent '#include' specifies the same file, and the macro in the '#ifndef' is already defined, then the file is entirely skipped, without even reading it.

There is also an explicit directive to tell the preprocessor that it need not include a file more than once. This is called '#pragma once', and was used *in addition to* the '#ifndef' conditional around the contents of the header file. '#pragma once' is now obsolete and should not be used at all.

In the Objective C language, there is a variant of '#include' called '#import' which includes a file, but does so at most once. If you use '#import' *instead of* '#include', then you don't need the conditionals inside the header file to prevent multiple execution of the contents.

'#import' is obsolete because it is not a well designed feature. It requires the users of a header file—the applications programmers—to know that a certain header file should only be included once. It is much better for the header file's implementor to write the file so that users don't need to know this. Using '#ifndef' accomplishes this goal.

1.3.5 Inheritance and Header Files

Inheritance is what happens when one object or file derives some of its contents by virtual copying from another object or file. In the case of C header files, inheritance means that one header file includes another header file and then replaces or adds something.

If the inheriting header file and the base header file have different names, then inheritance is straightforward: simply write '#include "*base*"' in the inheriting file.

Sometimes it is necessary to give the inheriting file the same name as the base file. This is less straightforward.

For example, suppose an application program uses the system header file 'sys/signal.h', but the version of '/usr/include/sys/signal.h' on a particular system doesn't do what the application program expects. It might be convenient to define a "local" version, perhaps under the name '/usr/local/include/sys/signal.h', to override or add to the one supplied by the system.

You can do this by using the option '-I.' for compilation, and writing a file 'sys/signal.h' that does what the application program expects. But making this file include the standard 'sys/signal.h' is not so easy—

writing '#include <sys/signal.h>' in that file doesn't work, because it includes your own version of the file, not the standard system version. Used in that file itself, this leads to an infinite recursion and a fatal error in compilation.

'#include </usr/include/sys/signal.h>' would find the proper file, but that is not clean, since it makes an assumption about where the system header file is found. This is bad for maintenance, since it means that any change in where the system's header files are kept requires a change somewhere else.

The clean way to solve this problem is to use '#include_next', which means, "Include the *next* file with this name." This directive works like '#include' except in searching for the specified file: it starts searching the list of header file directories *after* the directory in which the current file was found.

Suppose you specify '-I /usr/local/include', and the list of directories to search also includes '/usr/include'; and suppose that both directories contain a file named 'sys/signal.h'. Ordinary '#include <sys/signal.h>' finds the file under '/usr/local/include'. If that file contains '#include_next <sys/signal.h>', it starts searching after that directory, and finds the file in '/usr/include'.

1.4 Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

1.4.1 Simple Macros

A *simple macro* is a kind of abbreviation. It is a name which stands for a fragment of code. Some people refer to these as *manifest constants*.

Before you can use a macro, you must *define* it explicitly with the '#define' directive. '#define' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named 'BUFFER_SIZE' as an abbreviation for the text '1020'. If somewhere after this '#define' directive there comes a C statement of the form

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro 'BUFFER_SIZE', resulting in

```
foo = (char *) xmalloc (1020);
```

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessing directives. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. This is because it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (But if it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces as output

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name '`TABLESIZE`' when used in the program would go through two stages of expansion, resulting ultimately in '`1020`'.

This is not at all the same as defining '`TABLESIZE`' to be '`1020`'. The '`#define`' for '`TABLESIZE`' uses exactly the body you specify—in this case, '`BUFSIZE`'—and does not check to see whether it too is the name of a macro. It's only when you *use* '`TABLESIZE`' that the result of its expan-

sion is checked for more macro names. See Section 1.4.8.7 “Cascaded Macros,” page 27.

1.4.2 Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept *arguments*. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition. A macro that accepts arguments is called a *function-like macro* because the syntax for using it looks like a function call.

To define a macro that uses arguments, you write a ‘`#define`’ directive with a list of *argument names* in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a “minimum” macro in GNU C. See Section 1.4.8.4 “Side Effects,” page 23, for more information.)

To use a macro that expects arguments, you write the name of the macro followed by a list of *actual arguments* in parentheses, separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro ‘`min`’ include ‘`min (1, 2)`’ and ‘`min (x + 28, *p)`’.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro ‘`min`’ defined above, ‘`min (1, 2)`’ expands into

```
((1) < (2) ? (1) : (2))
```

where ‘`1`’ has been substituted for ‘`X`’ and ‘`2`’ for ‘`Y`’.

Likewise, ‘`min (x + 28, *p)`’ expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to `macro`: '`array[x = y`' and '`x + 1]`'. If you want to supply '`array[x = y, x + 1]`' as an argument, you must write it as '`array[(x = y, x + 1)]`', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, '`min (min (a, b), c)`' expands into this text:

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If a macro `foo` takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses, like this: '`foo ( )`'. Just '`foo ()`' is providing no arguments, which is an error if `foo` expects an argument. But '`foo0 ()`' is the correct way to call a macro defined to take zero arguments, like this:

```
#define foo0() ...
```

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named '`min`' in the same source file that defines the macro. If you write '`&min`' with no argument list, you refer to the function. If you write '`min (x, bb)`', with an argument list, the macro is expanded. If you write '`(min) (a, bb)`', where the name '`min`' is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function '`min`'.

You may not define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines 'FOO' to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define BAR (x) - 1 / (x)
```

(which defines 'BAR' to take no argument and always expand into '(x) - 1 / (x)').

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

1.4.3 Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

1.4.3.1 Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double underscores. Those preceding __GNUC__ in this table are standardized by ANSI C; the rest are GNU C extensions.

__FILE__
: This macro expands to the name of the current input file, in the form of a C string constant. The precise name returned is the one that was specified in '#include' or as the input file name argument.

__LINE__
: This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

  This and '__FILE__' are useful in generating an error message to report an inconsistency detected by the program; the

message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
                 "negative string length "
                 "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

A '#include' directive changes the expansions of '__FILE__' and '__LINE__' to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the '#include' directive, the expansions of '__FILE__' and '__LINE__' revert to the values they had before the '#include' (but '__LINE__' is then incremented by one as processing moves to the line after the '#include').

The expansions of both '__FILE__' and '__LINE__' are altered if a '#line' directive is used. See Section 1.6 "Combining Sources," page 36.

`__INCLUDE_LEVEL__`
: This macro expands to a decimal integer constant that represents the depth of nesting in include files. The value of this macro is incremented on every '#include' directive and decremented at every end of file. For input files specified by command line arguments, the nesting level is zero.

`__DATE__`
: This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like '"Jan 29 1987"' or '"Apr 1 1905"'.

`__TIME__`
: This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like '"23:59:01"'.

`__STDC__`
: This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

`__STDC_VERSION__`
: This macro expands to the C Standard's version number, a long integer constant of the form '*yyyymm*L' where *yyyy* and *mm* are the year and month of the Standard version. This signifies which version of the C Standard the preprocessor conforms to. Like '__STDC__', whether this version number is accurate for the entire implementation depends on what C compiler will operate on the output from the preprocessor.

`__GNUC__`
: This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if

you invoke the preprocessor directly, '__GNUC__' is undefined. The value identifies the major version number of GNU CC ('1' for GNU CC version 1, which is now obsolete, and '2' for version 2).

__GNUG__
: The GNU C compiler defines this when the compilation language is C++; use '__GNUG__' to distinguish between GNU C and GNU C++.

__cplusplus
: The draft ANSI standard for C++ used to require predefining this variable. Though it is no longer required, GNU C++ continues to define it, as do other popular C++ compilers. You can use '__cplusplus' to test whether a header is compiled by a C compiler or a C++ compiler.

__STRICT_ANSI__
: This macro is defined if and only if the '-ansi' switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

__BASE_FILE__
: This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

__VERSION__
: This macro expands to a string which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as '"2.6.0"'. The only reasonable use of this macro is to incorporate it into a string constant.

__OPTIMIZE__
: This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It is unwise to refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

__CHAR_UNSIGNED__
: This macro is defined if and only if the data type char is unsigned on the target machine. It exists to cause the standard header file 'limit.h' to work correctly. It is bad practice to refer to this macro yourself; instead, refer to the standard macros defined in 'limit.h'. The preprocessor uses this

macro to determine whether or not to sign-extend large character constants written in octal; see Section 1.5.2.1 "The '#if' Directive," page 29.

`__REGISTER_PREFIX__`
: This macro expands to a string describing the prefix applied to cpu registers in assembler code. It can be used to write assembler code that is usable in multiple environments. For example, in the 'm68k-aout' environment it expands to the string '""', but in the 'm68k-coff' environment it expands to the string '"%"'.

`__USER_LABEL_PREFIX__`
: This macro expands to a string describing the prefix applied to user generated labels in assembler code. It can be used to write assembler code that is usable in multiple environments. For example, in the 'm68k-aout' environment it expands to the string '"_"', but in the 'm68k-coff' environment it expands to the string '""'.

1.4.3.2 Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones. You can use 'cpp -dM' to see the values of predefined macros; see Section 1.9 "Invocation," page 38.

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example,

`unix`
: 'unix' is normally predefined on all Unix systems.

`BSD`
: 'BSD' is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

`vax`
: 'vax' is predefined on Vax computers.

`mc68000`
: 'mc68000' is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

`m68k`
: 'm68k' is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use 'mc68000' and some use 'm68k'. Some predefine both names. What happens in GNU C depends on the system you are using it on.

`M68020` ‘M68020’ has been observed to be predefined on some systems that use 68020 CPUs—in addition to ‘mc68000’ and ‘m68k’, which are less specific.

`_AM29K`
`_AM29000` Both ‘_AM29K’ and ‘_AM29000’ are predefined for the AMD 29000 CPU family.

`ns32000` ‘ns32000’ is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

`sun` ‘sun’ is predefined on all models of Sun computers.

`pyr` ‘pyr’ is predefined on all models of Pyramid computers.

`sequent` ‘sequent’ is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. Therefore, the option ‘-ansi’ inhibits the definition of these symbols.

This tends to make ‘-ansi’ useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with ‘-ansi’. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it will run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding ‘__’ at the beginning and end. Thus, the symbol `__vax__` would be available on a Vax, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled (when `cpp` is itself compiled) by the macro ‘CPP_PREDEFINES’, which should be a string containing ‘-D’ options, separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

This macro is usually specified in ‘tm.h’.

1.4.4 Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying '`foo (z)`' results in '`"foo (z)"`'.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character '`#`' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no '`#`'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

Here the actual argument for '`EXP`' is substituted once as given, into the '`if`' statement, and once as stringified, into the argument to '`fprintf`'. The '`do`' and '`while (0)`' are a kludge to make it possible to write '`WARN_IF (`*arg*`);`', which the resemblance of '`WARN_IF`' to a function would make C programmers want to do; see Section 1.4.8.3 "Swallow Semicolon," page 22.

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies the actual value of '`EXP`' into a separate string constant, resulting in text like

```
do { if (x == 0) \
        fprintf (stderr, "Warning: " "x == 0" "\n"); } \
while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) \
        fprintf (stderr, "Warning: x == 0\n"); } \
while (0)
```

Stringification in C involves more than putting doublequote characters around the fragment; it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying '`p = "foo\n";`' results in '`"p = \"foo\\n\";"`'.

However, backslashes that are not inside of string or character constants are not duplicated: '`\n`' by itself stringifies to '`"\n"`'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

1.4.5 Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator '`##`' in the macro body. When the macro is called, after actual arguments are substituted, all '`##`' operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the '`##`'.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with '`_command`'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as '`1.5`' and '`e3`') into a number. Also, multi-character operators such as '`+=`' can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with '`x`' on one side and '`+`' on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the '`x`' and '`+`' side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating '`/`' and '`*`': the '`/*`' sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. Also, you can freely use comments next to a '`##`' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

1.4.6 Undefining Macros

To *undefine* a macro means to cancel its definition. This is done with the '`#undef`' directive. '`#undef`' is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;

x = FOO;
```

In this example, 'FOO' had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of '#undef' directive will cancel definitions with arguments or definitions that don't expect arguments. The '#undef' directive has no effect when used on a name not currently defined as a macro.

1.4.7 Redefining Macros

Redefining a macro means defining (with '#define') a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see Section 1.3 "Header Files," page 3), so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with '#undef' before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

1.4.8 Pitfalls and Subtleties of Macros

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counterintuitive consequences that you must watch out for.

1.4.8.1 Improperly Nested Constructs

Recall that when a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls.

It is possible to piece together a macro call coming partially from the macro body and partially from the actual arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand '`call_with_1 (double)`' into '`(2*(1))`'.

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)
```

This bizarre example expands to '`fprintf (stderr, "%s %d", p, 35)`'!

1.4.8.2 Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name had parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. Here is why it is best to write macros that way.

Suppose you define a macro as follows,

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many '`int`' objects are needed to hold a certain number of '`char`' objects.) Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

However, unintended grouping can result in another way. Consider 'sizeof ceil_div(1, 2)'. That has the appearance of a C expression that would compute the size of the type of 'ceil_div (1, 2)', but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the 'sizeof' when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here, then, is the recommended way to define 'ceil_div':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

1.4.8.3 Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument 'p' says where to find it) across whitespace characters:

```
#define SKIP_SPACES (p, limit)  \
{ register char *lim = (limit); \
  while (p != lim) {            \
    if (*p++ != ' ') {          \
      p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would be laid out if not part of a macro definition.

A call to this macro might be 'SKIP_SPACES (p, lim)'. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in 'SKIP_SPACES (p, lim);'

But this can cause trouble before 'else' statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

The presence of two statements—the compound statement and a null statement—in between the 'if' condition and the 'else' makes invalid C code.

The definition of the macro 'SKIP_SPACES' can be altered to solve this problem, using a 'do ... while' statement. Here is how:

```
#define SKIP_SPACES (p, limit)     \
do { register char *lim = (limit); \
     while (p != lim) {            \
       if (*p++ != ' ') {          \
         p--; break; }}}           \
while (0)
```

Now 'SKIP_SPACES (p, lim);' expands into

```
do {...} while (0);
```

which is one statement.

1.4.8.4 Duplication of Side Effects

Many C programs define a macro 'min', for "minimum", like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where 'x + y' has been substituted for 'X' and 'foo (z)' for 'Y'.

The function 'foo' is used only once in the statement as it appears in the program, but the expression 'foo (z)' has been substituted twice into the macro expansion. As a result, 'foo' might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. We say that 'min' is an *unsafe* macro.

The best solution to this problem is to define 'min' in a way that computes the value of 'foo (z)' only once. The C language offers no standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                      \
({ typeof (X) __x = (X), __y = (Y);   \
   (__x < __y) ? __x : __y; })
```

If you do not wish to use GNU C extensions, the only solution is to be careful when *using* the macro 'min'. For example, you can calculate the value of 'foo (z)', save it in a variable, and use that variable in 'min':

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
```

```
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where we assume that '`foo`' returns type '`int`').

1.4.8.5 Self-Referential Macros

A *self-referential* macro is one whose name appears in its definition. A special feature of ANSI Standard C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

```
#define foo (4 + foo)
```

where '`foo`' is also a variable in your program.

Following the ordinary rules, each reference to '`foo`' will expand into '`(4 + foo)`'; then this will be rescanned and will expand into '`(4 + (4 + foo))`'; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at '`(4 + foo)`'. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of '`foo`' wherever '`foo`' is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that '`foo`' is a variable will not expect that it is a macro as well. The reader will come across the identifier '`foo`' in the program and think its value should be that of the variable '`foo`', whereas in fact the value is four greater.

The special rule for self-reference applies also to *indirect* self-reference. This is the case where a macro `x` expands to use a macro '`y`', and the expansion of '`y`' refers to the macro '`x`'. The resulting reference to '`x`' comes indirectly from the expansion of '`x`', so it is a self-reference and is not further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

'`x`' would expand into '`(4 + (2 * x))`'. Clear?

But suppose '`y`' is used elsewhere, not from the definition of '`x`'. Then the use of '`x`' in the expansion of '`y`' is not a self-reference because '`x`' is not "in progress". So it does expand. However, the expansion of '`x`' contains a reference to '`y`', and that is an indirect self-reference now because '`y`' is "in progress". The result is that '`y`' expands to '`(2 * (4 + y))`'.

It is not clear that this behavior would ever be useful, but it is specified by the ANSI C standard, so you may need to understand it.

1.4.8.6 Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the actual arguments are scanned *twice* to expand macro calls in them.

Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro (see Section 1.4.8.5 "Self-Reference," page 24): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

The prescan is not done when an argument is stringified or concatenated. Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to '`"foo"`'. Once more, prescan has been prevented from having any noticeable effect.

More precisely, stringification and concatenation use the argument as written, in un-prescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.

```
#define str(s) #s lose(s)
#define foo 4
str (foo)
```

expands to '`"foo" lose(4)`'.

You might now ask, "Why mention the prescan, if it makes no difference? And why not skip it and make the preprocessor faster?" The answer is that the prescan does make a difference in three special cases:

- Nested calls to a macro.
- Macros that call other macros that stringify or concatenate.
- Macros whose expansions contain unshielded commas.

We say that *nested* calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if 'f' is a macro that expects one argument, 'f (f (1))' is a nested pair of calls to 'f'. The desired expansion is made by expanding 'f (1)' and substituting that into the definition of 'f'. The prescan causes the expected result to happen. Without the prescan, 'f (1)' itself would be substituted as an actual argument, and the inner use of 'f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. Here is an example:

```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))

bar(foo)
```

We would like 'bar(foo)' to turn into '(1 + (foo))', which would then turn into '(1 + (a,b))'. But instead, 'bar(foo)' expands into 'lose(a,b)', and you get an error because lose requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
#define bar(x) lose((x))
```

The problem is more serious when the operands of the macro are not expressions; for example, when they are statements. Then parentheses are unacceptable because they would make for invalid C code:

```
#define foo { int a, b; ... }
```

In GNU C you can shield the commas using the '({...})' construct which turns a compound statement into an expression:

```
#define foo ({ int a, b; ... })
```

Or you can rewrite the macro definition to avoid such commas:

```
#define foo { int a; int b; ... }
```

There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it—if you use two levels of macros. Let's add a new macro 'xstr' to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands into '"4"', not '"foo"'. The reason for the difference is that the argument of 'xstr' is expanded at prescan (because 'xstr' does not specify stringification or concatenation of the argument). The result of prescan then forms the actual argument for 'str'. 'str' uses its argument without prescan because it performs stringification; but it cannot prevent or undo the prescanning already done by 'xstr'.

1.4.8.7 Cascaded Use of Macros

A *cascade* of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This is not at all the same as defining 'TABLESIZE' to be '1020'. The '#define' for 'TABLESIZE' uses exactly the body you specify—in this case, 'BUFSIZE'—and does not check to see whether it too is the name of a macro.

It's only when you *use* 'TABLESIZE' that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of 'BUFSIZE' at some point in the source file. 'TABLESIZE', defined as shown, will always expand using the definition of 'BUFSIZE' that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now 'TABLESIZE' expands (in two stages) to '37'. (The '#undef' is to prevent any warning about the nontrivial redefinition of BUFSIZE.)

1.4.9 Newlines in Macro Arguments

Traditional macro processing carries forward all newlines in macro arguments into the expansion of the macro. This means that, if some of the arguments are substituted more than once, or not at all, or out

of order, newlines can be duplicated, lost, or moved around within the expansion. If the expansion consists of multiple statements, then the effect is to distort the line numbers of some of these statements. The result can be incorrect line numbers, in error messages or displayed in a debugger.

The GNU C preprocessor operating in ANSI C mode adjusts appropriately for multiple use of an argument—the first use expands all the newlines, and subsequent uses of the same argument produce no newlines. But even in this mode, it can produce incorrect line numbering if arguments are used out of order, or not used at all.

Here is an example illustrating this problem:

```
#define ignore_second_arg(a,b,c) a; c

ignore_second_arg (foo (),
                   ignored (),
                   syntax error);
```

The syntax error triggered by the tokens '`syntax error`' results in an error message citing line four, even though the statement text comes from line five.

1.5 Conditionals

In a macro processor, a *conditional* is a directive that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an '`if`' statement in C, but it is important to understand the difference between them. The condition in an '`if`' statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessing conditional directive is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

1.5.1 Why Conditionals are Used

Generally there are three kinds of reason to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to library routines that do not exist on

the other system. When this happens, it is not enough to avoid executing the invalid code: merely having it in the program makes it impossible to link the program and run it. With a preprocessing conditional, the offending code can be effectively excised from the program when it is not valid.

- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data, or prints the values of those data for debugging, while the other does not.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it as a sort of comment for future reference.

Most simple programs that are intended to run on only one machine will not need to use preprocessing conditionals.

1.5.2 Syntax of Conditionals

A conditional in the C preprocessor begins with a *conditional directive*: '`#if`', '`#ifdef`' or '`#ifndef`'. See Section 1.5.4 "Conditionals-Macros," page 32, for information on '`#ifdef`' and '`#ifndef`'; only '`#if`' is explained here.

1.5.2.1 The '#if' Directive

The '`#if`' directive in its simplest form consists of

```
#if expression
controlled text
#endif /* expression */
```

The comment following the '`#endif`' is not required, but it is a good practice because it helps people match the '`#endif`' to the corresponding '`#if`'. Such comments should always be used, except in short conditionals that are not nested. In fact, you can put anything at all after the '`#endif`' and it will be ignored by the GNU C preprocessor, but only comments are acceptable in ANSI Standard C.

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants, which are all regarded as `long` or `unsigned long`.
- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type '`char`' for these character constants; therefore,

whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats 'char' as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.

- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations ('&&' and '||').
- Identifiers that are not macros, which are all treated as zero(!).
- Macro calls. All macro calls in the expression are expanded before actual computation of the expression's value begins.

Note that 'sizeof' operators and enum-type values are not allowed. enum-type values, like all other identifiers that are not taken as macro calls and expanded, are treated as zero.

The *controlled text* inside of a conditional can include preprocessing directives. Then the directives inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the '#if' and '#endif' directives must balance.

1.5.2.2 The '#else' Directive

The '#else' directive can be added to a conditional to provide alternative text to be used if the condition is false. This is what it looks like:

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If *expression* is nonzero, and thus the *text-if-true* is active, then '#else' acts like a failing conditional and the *text-if-false* is ignored. Contrariwise, if the '#if' conditional fails, the *text-if-false* is considered included.

1.5.2.3 The '#elif' Directive

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
...
#else /* X != 1 */
```

```
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional directive, '#elif', allows this to be abbreviated as follows:

```
#if X == 1
...
#elif X == 2
...
#else /* X != 2 and X != 1*/
...
#endif /* X != 2 and X != 1*/
```

'#elif' stands for "else if". Like '#else', it goes in the middle of a '#if'-'#endif' pair and subdivides it; it does not require a matching '#endif' of its own. Like '#if', the '#elif' directive includes an expression to be tested.

The text following the '#elif' is processed only if the original '#if'-condition failed and the '#elif' condition succeeds. More than one '#elif' can go in the same '#if'-'#endif' group. Then the text after each '#elif' is processed only if the '#elif' condition succeeds after the original '#if' and any previous '#elif' directives within it have failed. '#else' is equivalent to '#elif 1', and '#else' is allowed after any number of '#elif' directives, but '#elif' may not follow '#else'.

1.5.3 Keeping Deleted Code for Future Reference

If you replace or delete a part of the program but want to keep the old code around as a comment for future reference, the easy way to do this is to put '#if 0' before it and '#endif' after it. This is better than using comment delimiters '/*' and '*/' since those won't work if the code already contains comments (C comments do not nest).

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced '#if' and '#endif').

Conversely, do not use '#if 0' for comments which are not C code. Use the comment delimiters '/*' and '*/' instead. The interior of '#if 0' must consist of complete tokens; in particular, singlequote characters must balance. But comments often contain unbalanced singlequote characters (known in English as apostrophes). These confuse '#if 0'. They do not confuse '/*'.

1.5.4 Conditionals and Macros

Conditionals are useful in connection with macros or assertions, because those are the only ways that an expression's value can vary from one compilation to another. A '#if' directive whose expression uses no macros or assertions is equivalent to '#if 1' or '#if 0'; you might as well determine which one, by computing the value of the expression yourself, and then simplify the program.

For example, here is a conditional that tests the expression 'BUFSIZE == 1020', where 'BUFSIZE' must be a macro.

```
#if BUFSIZE == 1020
  printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

(Programmers often wish they could test the size of a variable or data type in '#if', but this does not work. The preprocessor does not understand sizeof, or typedef names, or even the type keywords such as int.)

The special operator 'defined' is used in '#if' expressions to test whether a certain name is defined as a macro. Either 'defined *name*' or 'defined (*name*)' is an expression whose value is 1 if *name* is defined as macro at the current point in the program, and 0 otherwise. For the 'defined' operator it makes no difference what the definition of the macro is; all that matters is whether there is a definition. Thus, for example,

```
#if defined (vax) || defined (ns16000)
```

would succeed if either of the names 'vax' and 'ns16000' is defined as a macro. You can test the same condition using assertions (see Section 1.5.5 "Assertions," page 33), like this:

```
#if #cpu (vax) || #cpu (ns16000)
```

If a macro is defined and later undefined with '#undef', subsequent use of the 'defined' operator returns 0, because the name is no longer defined. If the macro is defined again with another '#define', 'defined' will recommence returning 1.

Conditionals that test whether just one name is defined are very common, so there are two special short conditional directives for this case.

#ifdef *name*
: is equivalent to '#if defined (*name*)'.

#ifndef *name*
: is equivalent to '#if ! defined (*name*)'.

Macro definitions can vary between compilations for several reasons.

- Some macros are predefined on each kind of machine. For example, on a Vax, the name '`vax`' is a predefined macro. On other machines, it would not be defined.
- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros are a common way of allowing users to customize a program for different machines or applications. For example, the macro '`BUFSIZE`' might be defined in a configuration file for your program that is included as a header file in each source file. You would use '`BUFSIZE`' in a preprocessing conditional in order to generate different code depending on the chosen configuration.
- Macros can be defined or undefined with '`-D`' and '`-U`' command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See Section 1.9 "Invocation," page 38.

1.5.5 Assertions

Assertions are a more systematic alternative to macros in writing conditionals to test what sort of computer or system the compiled program will run on. Assertions are usually predefined, but you can define them with preprocessing directives or command-line options.

The macros traditionally used to describe the type of target are not classified in any way according to which question they answer; they may indicate a hardware architecture, a particular hardware model, an operating system, a particular version of an operating system, or specific configuration options. These are jumbled together in a single namespace. In contrast, each assertion consists of a named question and an answer. The question is usually called the *predicate*. An assertion looks like this:

```
#predicate (answer)
```

You must use a properly formed identifier for *`predicate`*. The value of *`answer`* can be any sequence of words; all characters are significant except for leading and trailing whitespace, and differences in internal whitespace sequences are ignored. Thus, '`x + y`' is different from '`x+y`' but equivalent to '`x + y`'. '`)`' is not allowed in an answer.

Here is a conditional to test whether the answer *`answer`* is asserted for the predicate *`predicate`*:

```
#if #predicate (answer)
```

There may be more than one answer asserted for a given predicate. If you omit the answer, you can test whether *any* answer is asserted for `predicate`:

```
#if #predicate
```

Most of the time, the assertions you test will be predefined assertions. GNU C provides three predefined predicates: `system`, `cpu`, and `machine`. `system` is for assertions about the type of software, `cpu` describes the type of computer architecture, and `machine` gives more information about the computer. For example, on a GNU system, the following assertions would be true:

```
#system (gnu)
#system (mach)
#system (mach 3)
#system (mach 3.subversion)
#system (hurd)
#system (hurd version)
```

and perhaps others. The alternatives with more or less version information let you ask more or less detailed questions about the type of system software.

On a Unix system, you would find `#system (unix)` and perhaps one of: `#system (aix)`, `#system (bsd)`, `#system (hpux)`, `#system (lynx)`, `#system (mach)`, `#system (posix)`, `#system (svr3)`, `#system (svr4)`, or `#system (xpg4)` with possible version numbers following.

Other values for `system` are `#system (mvs)` and `#system (vms)`.

Portability note: Many Unix C compilers provide only one answer for the `system` assertion: `#system (unix)`, if they support assertions at all. This is less than useful.

An assertion with a multi-word answer is completely different from several assertions with individual single-word answers. For example, the presence of `system (mach 3.0)` does not mean that `system (3.0)` is true. It also does not directly imply `system (mach)`, but in GNU C, that last will normally be asserted as well.

The current list of possible assertion values for `cpu` is: `#cpu (a29k)`, `#cpu (alpha)`, `#cpu (arm)`, `#cpu (clipper)`, `#cpu (convex)`, `#cpu (elxsi)`, `#cpu (tron)`, `#cpu (h8300)`, `#cpu (i370)`, `#cpu (i386)`, `#cpu (i860)`, `#cpu (i960)`, `#cpu (m68k)`, `#cpu (m88k)`, `#cpu (mips)`, `#cpu (ns32k)`, `#cpu (hppa)`, `#cpu (pyr)`, `#cpu (ibm032)`, `#cpu (rs6000)`, `#cpu (sh)`, `#cpu (sparc)`, `#cpu (spur)`, `#cpu (tahoe)`, `#cpu (vax)`, `#cpu (we32000)`.

You can create assertions within a C program using '#assert', like this:

```
#assert predicate (answer)
```

(Note the absence of a '#' before *predicate*.)

Each time you do this, you assert a new true answer for *predicate*. Asserting one answer does not invalidate previously asserted answers; they all remain true. The only way to remove an assertion is with '#unassert'. '#unassert' has the same syntax as '#assert'. You can also remove all assertions about *predicate* like this:

```
#unassert predicate
```

You can also add or cancel assertions using command options when you run gcc or cpp. See Section 1.9 "Invocation," page 38.

1.5.6 The '#error' and '#warning' Directives

The directive '#error' causes the preprocessor to report a fatal error. The rest of the line that follows '#error' is used as the error message.

You would use '#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
#ifdef __vax__
#error Won't work on Vaxen.  See comments at get_last_object.
#endif
```

See Section 1.4.3.2 "Nonstandard Predefined," page 15, for why this works.

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with '#error'. For example,

```
#if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
    || HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE should not be divisible by a small prime
#endif
```

The directive '#warning' is like the directive '#error', but causes the preprocessor to issue a warning and continue preprocessing. The rest of the line that follows '#warning' is used as the warning message.

You might use '#warning' in obsolete header files, with a message directing the user to the header file which should be used instead.

1.6 Combining Source Files

One of the jobs of the C preprocessor is to inform the C compiler of where each line of C code came from: which source file and which line number.

C code can come from multiple source files if you use '`#include`'; both '`#include`' and the use of conditionals and macros can cause the line number of a line in the preprocessor output to be different from the line's number in the original source file. You will appreciate the value of making both the C compiler (in error messages) and symbolic debuggers such as GDB use the line numbers in your source file.

The C preprocessor builds on this feature by offering a directive by which you can control the feature explicitly. This is useful when a file for input to the C preprocessor is the output from another program such as the `bison` parser generator, which operates on another file that is the true source file. Parts of the output from `bison` are generated from scratch, other parts come from a standard parser file. The rest are copied nearly verbatim from the source file, but their line numbers in the `bison` output are not the same as their original line numbers. Naturally you would like compiler error messages and symbolic debuggers to know the original source file and line number of each line in the `bison` input.

`bison` arranges this by writing '`#line`' directives into the output file. '`#line`' is a directive that specifies the original line number and source file name for subsequent input in the current preprocessor input file. '`#line`' has three variants:

`#line` *`linenum`*

Here *`linenum`* is a decimal integer constant. This specifies that the line number of the following line of input, in its original source file, was *`linenum`*.

`#line` *`linenum filename`*

Here *`linenum`* is a decimal integer constant and *`filename`* is a string constant. This specifies that the following line of input came originally from source file *`filename`* and its line number there was *`linenum`*. Keep in mind that *`filename`* is not just a file name; it is surrounded by doublequote characters so that it looks like a string constant.

`#line` *`anything else`*

`anything else` is checked for macro calls, which are expanded. The result should be a decimal integer constant followed optionally by a string constant, as described above.

'#line' directives alter the results of the '__FILE__' and '__LINE__' predefined macros from that point on. See Section 1.4.3.1 "Standard Predefined," page 12.

The output of the preprocessor (which is the input for the rest of the compiler) contains directives that look much like '#line' directives. They start with just '#' instead of '#line', but this is followed by a line number and file name as in '#line'. See Section 1.8 "Output," page 37.

1.7 Miscellaneous Preprocessing Directives

This section describes three additional preprocessing directives. They are not very useful, but are mentioned for completeness.

The *null directive* consists of a '#' followed by a Newline, with only whitespace (including comments) in between. A null directive is understood as a preprocessing directive but has no effect on the preprocessor output. The primary significance of the existence of the null directive is that an input line consisting of just a '#' will produce no output, rather than a line of output containing just a '#'. Supposedly some old C programs contain such lines.

The ANSI standard specifies that the '#pragma' directive has an arbitrary, implementation-defined effect. In the GNU C preprocessor, '#pragma' directives are not used, except for '#pragma once' (see Section 1.3.4 "Once-Only," page 6). However, they are left in the preprocessor output, so they are available to the compilation pass.

The '#ident' directive is supported for compatibility with certain other systems. It is followed by a line of text. On some systems, the text is copied into a special place in the object file; on most systems, the text is ignored and this directive has no effect. Typically '#ident' is only used in header files supplied with those systems where it is meaningful.

1.8 C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessing directive lines have been replaced with blank lines and all comments with spaces. Whitespace within a line is not altered; however, a space is inserted after the expansions of most macro calls.

Source file name and line number information is conveyed by lines of the form

```
# linenum filename flags
```

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file *filename* at line *linenum*.

After the file name comes zero or more flags, which are '`1`', '`2`' or '`3`'. If there are multiple flags, spaces separate them. Here is what the flags mean:

'`1`' This indicates the start of a new file.

'`2`' This indicates returning to a file (after having included another file).

'`3`' This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

1.9 Invoking the C Preprocessor

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful on its own.

The C preprocessor expects two file names as arguments, *`infile`* and *`outfile`*. The preprocessor reads *`infile`* together with any other files it specifies with '`#include`'. All the output generated by the combined input files is written in *`outfile`*.

Either *`infile`* or *`outfile`* may be '`-`', which as *`infile`* means to read from standard input and as *`outfile`* means to write to standard output. Also, if *`outfile`* or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here is a table of command options accepted by the C preprocessor. These options can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

'`-P`' Inhibit generation of '`#`'-lines with line-number information in the output from the preprocessor (see Section 1.8 "Output," page 37). This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the '`#`'-lines.

'`-C`' Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

'`-traditional`'
Try to imitate the behavior of old-fashioned C, as opposed to ANSI C.

- Traditional macro expansion pays no attention to singlequote or doublequote characters; macro argument sym-

bols are replaced by the argument values even when they appear within apparent string or character constants.

- Traditionally, it is permissible for a macro expansion to end in the middle of a string or character constant. The constant continues into the text surrounding the macro call.
- However, traditionally the end of the line terminates a string or character constant, with no error.
- In traditional C, a comment is equivalent to no text at all. (In ANSI C, a comment counts as whitespace.)
- Traditional C does not have the concept of a "preprocessing number". It considers '`1.0e+4`' to be three tokens: '`1.0e`', '`+`', and '`4`'.
- A macro is not suppressed within its own definition, in traditional C. Thus, any macro that is used recursively inevitably causes an error.
- The character '`#`' has no special meaning within a macro definition in traditional C.
- In traditional C, the text at the end of a macro expansion can run together with the text after the macro call, to produce a single token. (This is impossible in ANSI C.)
- Traditionally, '\' inside a macro argument suppresses the syntactic significance of the following character.

'`-trigraphs`'
: Process ANSI standard trigraph sequences. These are three-character sequences, all starting with '`??`', that are defined by ANSI C to stand for single characters. For example, '`??/`' stands for '\', so '`'??/n'`' is a character constant for a newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless '`-trigraphs`' is used, but if you ever notice the difference it will be with relief.

  You don't want to know any more about trigraphs.

'`-pedantic`'
: Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows '`#else`' or '`#endif`'.

'`-pedantic-errors`'
: Like '`-pedantic`', except that errors are produced rather than warnings.

'-Wtrigraphs'
: Warn if any trigraphs are encountered (assuming they are enabled).

'-Wcomment'
: Warn whenever a comment-start sequence '/*' appears in a comment.

'-Wall'
: Requests both '-Wtrigraphs' and '-Wcomment' (but not '-Wtraditional').

'-Wtraditional'
: Warn about certain constructs that behave differently in traditional and ANSI C.

'-I *directory*'
: Add the directory *directory* to the end of the list of directories to be searched for header files (see Section 1.3.2 "Include Syntax," page 4). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

'-I-'
: Any directories specified with '-I' options before the '-I-' option are searched only for the case of '#include "*file*"'; they are not searched for '#include <*file*>'.

  If additional directories are specified with '-I' options after the '-I-', these directories are searched for all '#include' directives.

  In addition, the '-I-' option inhibits the use of the current directory as the first search directory for '#include "*file*"'. Therefore, the current directory is searched only if it is requested explicitly with '-I.'. Specifying both '-I-' and '-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

'-nostdinc'
: Do not search the standard system directories for header files. Only the directories you have specified with '-I' options (and the current directory, if appropriate) are searched.

'-nostdinc++'
: Do not search for header files in the C++-specific standard directories, but do still search the other standard directories. (This option is used when building libg++.)

'-D *name*'
: Predefine *name* as a macro, with definition '1'.

'-D *name*=*definition*'
Predefine *name* as a macro, with definition *definition*. There are no restrictions on the contents of *definition*, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. If you use more than one '-D' for the same *name*, the rightmost definition takes effect.

'-U *name*' Do not predefine *name*. If both '-U' and '-D' are specified for one name, the '-U' beats the '-D' and the name is not predefined.

'-undef' Do not predefine any nonstandard macros.

'-A *predicate*(*answer*)'
Make an assertion with the predicate *predicate* and answer *answer*. See Section 1.5.5 "Assertions," page 33.

You can use '-A-' to disable all predefined assertions; it also undefines all predefined macros that identify the type of target system.

'-dM' Instead of outputting the result of preprocessing, output a list of '#define' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor; assuming you have no file 'foo.h', the command

```
touch foo.h; cpp -dM foo.h
```

will show the values of any predefined macros.

'-dD' Like '-dM' except in two respects: it does *not* include the predefined macros, and it outputs *both* the '#define' directives and the result of preprocessing. Both kinds of output go to the standard output file.

'-M [-MG]' Instead of outputting the result of preprocessing, output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using '\'-newline.

'-MG' says to treat missing header files as generated files and assume they live in the same directory as the source file. It must be specified in addition to '-M'.

This feature is used in automatic updating of makefiles.

'-MM [-MG]'
: Like '-M' but mention only the files included with '#include "*file*"'. System header files included with '#include <*file*>' are omitted.

'-MD *file*'
: Like '-M' but the dependency information is written to *file*. This is in addition to compiling the file as specified—'-MD' does not inhibit ordinary compilation the way '-M' does.

  When invoking gcc, do not specify the *file* argument. Gcc will create file names made by replacing ".c" with ".d" at the end of the input file names.

  In Mach, you can use the utility md to merge multiple dependency files into a single dependency file suitable for using with the 'make' command.

'-MMD *file*'
: Like '-MD' except mention only user header files, not system header files.

'-H'
: Print the name of each header file used, in addition to other normal activities.

'-imacros *file*'
: Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of '-imacros *file*' is to make the macros defined in *file* available for use in the main input.

'-include *file*'
: Process *file* as input, and include all the resulting output, before processing the regular input file.

'-idirafter *dir*'
: Add the directory *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that '-I' adds to).

'-iprefix *prefix*'
: Specify *prefix* as the prefix for subsequent '-iwithprefix' options.

'-iwithprefix *dir*'
: Add a directory to the second include path. The directory's name is made by concatenating *prefix* and *dir*, where *prefix* was specified previously with '-iprefix'.

'-isystem *dir*'
: Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

'-lang-c'
'-lang-c++'
'-lang-objc'
'-lang-objc++'
: Specify the source language. '-lang-c++' makes the preprocessor handle C++ comment syntax (comments may begin with '//', in which case they end at end of line), and includes extra default include directories for C++; and '-lang-objc' enables the Objective C '#import' directive. '-lang-c' explicitly turns off both of these extensions, and '-lang-objc++' enables both.

  These options are generated by the compiler driver gcc, but not passed from the 'gcc' command line.

'-lint'
: Look for commands to the program checker lint embedded in comments, and emit them preceded by '#pragma lint'. For example, the comment '/* NOTREACHED */' becomes '#pragma lint NOTREACHED'.

  This option is available only when you call cpp directly; gcc will not pass it from its command line.

'-$'
: Forbid the use of '$' in identifiers. This is required for ANSI conformance. gcc automatically supplies this option to the preprocessor if you specify '-ansi', but gcc doesn't recognize the '-$' option itself—to use it without the other effects of '-ansi', you must call the preprocessor directly.

Concept Index

#

'##' . . . 18

A

arguments in macro definitions . . . 10
assertions . . . 33
assertions, undoing . . . 35

B

blank macro arguments . . . 11

C

cascaded macros . . . 27
commenting out code . . . 31
computed '#include' . . . 5
concatenation . . . 18
conditionals . . . 28

D

directives . . . 2

E

expansion of arguments . . . 25

F

function-like macro . . . 10

H

header file . . . 3

I

including just once . . . 6
inheritance . . . 7
invocation of the preprocessor . . . 38

L

line control . . . 36

M

macro argument expansion . . . 25
macro body uses macro . . . 27
macros with argument . . . 10
manifest constant . . . 8

N

newlines in macro arguments . . . 27
null directive . . . 37

O

options . . . 38
output format . . . 37
overriding a header file . . . 7

P

parentheses in macro bodies . . . 21
pitfalls of macros . . . 20
predefined macros . . . 12
predicates . . . 33
preprocessing directives . . . 2
prescan of macro arguments . . . 25
problems with macros . . . 20

R

redefining macros . . . 20
repeated inclusion . . . 6
retracting assertions . . . 35

S

second include path . . . 42
self-reference . . . 24
semicolons (after macro calls) . . . 22
side effects (in macro arguments) . . . 23
simple macro . . . 8
space as macro argument . . . 11
standard predefined macros . . . 12
stringification . . . 17

T

testing predicates . . . 33

U

unassert 35
undefining macros 19
unsafe macros 23

Index of Directives, Macros and Options

#

#assert 34
#cpu 34
#define 10
#elif 30
#else 30
#error 35
#ident 37
#if 29
#ifdef 32
#ifndef 32
#import 7
#include 4
#include_next 8
#line 36
#machine 34
#pragma 37
#pragma once 7
#system 34
#unassert 35
#warning 35

-

-$ 43
-A 41
-C 38
-D 40
-dD 41
-dM 41
-H 42
-I 40
-idirafter 42
-imacros 42
-include 42
-iprefix 42
-isystem 43
-iwithprefix 42
-lang-c 43
-lang-c++ 43
-lang-objc 43
-lang-objc++ 43
-M 41
-MD 42
-MM 42
-MMD 42
-nostdinc 40
-nostdinc++ 40
-P 38
-pedantic 39
-pedantic-errors 39
-traditional 38
-trigraphs 39
-U 41
-undef 41
-Wall 40
-Wcomment 40
-Wtraditional 40
-Wtrigraphs 40

_

__BASE_FILE__ 14
__CHAR_UNSIGNED__ 14
__cplusplus 14
__DATE__ 13
__FILE__ 12
__GNUC__ 13
__GNUG__ 14
__INCLUDE_LEVEL_ 13
__LINE__ 12
__OPTIMIZE__ 14
__REGISTER_PREFIX__ 15
__STDC__ 13
__STDC_VERSION__ 13
__STRICT_ANSI__ 14
__TIME__ 13
__USER_LABEL_PREFIX__ 15
__VERSION__ 14
_AM29000 16
_AM29K 16

B

BSD 15

D

defined 32

M

M68020 16
m68k 15
mc68000 15

N

ns32000 16

P

pyr 16

S

sequent 16
sun 16
system header files 3

U

unix 15

V

vax 15

Using as

The GNU Assembler

January 1994

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of `as` for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

Dean Elsner, Jay Fenlason & friends

Using `as`

Edited by Roland Pesch for Cygnus Support

Copyright © 1991, 1992, 1993, 1994 1995 Free Software Foundation, Inc.

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided 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 manual into another language, under the above conditions for modified versions.

Table of Contents

1 Overview 1
1.1 Structure of this Manual 4
1.2 as, the GNU Assembler 4
1.3 Object File Formats 5
1.4 Command Line 5
1.5 Input Files 5
1.6 Output (Object) File 6
1.7 Error and Warning Messages 6

2 Command-Line Options 9
2.1 Enable Listings: `-a[dhlns]` 9
2.2 `-D` 9
2.3 Work Faster: `-f` 10
2.4 `.include` search path: `-I` *path* 10
2.5 Difference Tables: `-K` 10
2.6 Include Local Labels: `-L` 10
2.7 Name the Object File: `-o` 11
2.8 Join Data and Text Sections: `-R` 11
2.9 Display Assembly Statistics: `--statistics` 11
2.10 Announce Version: `-v` 11
2.11 Suppress Warnings: `-W` 11
2.12 Generate Object File in Spite of Errors: `-Z` 12

3 Syntax 13
3.1 Preprocessing 13
3.2 Whitespace 13
3.3 Comments 14
3.4 Symbols 14
3.5 Statements 15
3.6 Constants 16
3.6.1 Character Constants 16
3.6.1.1 Strings 16
3.6.1.2 Characters 17
3.6.2 Number Constants 17
3.6.2.1 Integers 17
3.6.2.2 Bignums 18
3.6.2.3 Flonums 18

4 Sections and Relocation 21
4.1 Background 21

4.2 ld Sections 23
4.3 as Internal Sections 24
4.4 Sub-Sections 24
4.5 bss Section 25

5 Symbols 27

5.1 Labels 27
5.2 Giving Symbols Other Values 27
5.3 Symbol Names 27
5.4 The Special Dot Symbol 28
5.5 Symbol Attributes 29
5.5.1 Value 29
5.5.2 Type 29
5.5.3 Symbol Attributes: `a.out` 29
5.5.3.1 Descriptor 29
5.5.3.2 Other 30
5.5.4 Symbol Attributes for COFF 30
5.5.4.1 Primary Attributes 30
5.5.4.2 Auxiliary Attributes 30
5.5.5 Symbol Attributes for SOM 30

6 Expressions 31

6.1 Empty Expressions 31
6.2 Integer Expressions 31
6.2.1 Arguments 31
6.2.2 Operators 32
6.2.3 Prefix Operator 32
6.2.4 Infix Operators 32

7 Assembler Directives 35

7.1 `.abort` 35
7.2 `.ABORT` 35
7.3 `.align` *abs-expr* , *abs-expr* 35
7.4 `.app-file` *string* 36
7.5 `.ascii "`*string*`"` 36
7.6 `.asciz "`*string*`"` 36
7.7 `.byte` *expressions* 36
7.8 `.comm` *symbol* , *length* 36
7.9 `.data` *subsection* 36
7.10 `.def` *name* 37
7.11 `.desc` *symbol*, *abs-expression* 37
7.12 `.dim` 37
7.13 `.double` *flonums* 37
7.14 `.eject` 37

7.15 `.else` 37
7.16 `.endef` 38
7.17 `.endif` 38
7.18 `.equ` *symbol*, *expression* 38
7.19 `.extern` 38
7.20 `.file` *string* 38
7.21 `.fill` *repeat* , *size* , *value* 38
7.22 `.float` *flonums* 39
7.23 `.global` *symbol*, `.globl` *symbol* 39
7.24 `.hword` *expressions* 39
7.25 `.ident` 39
7.26 `.if` *absolute expression* 40
7.27 `.include "`*file*`"` 40
7.28 `.int` *expressions* 40
7.29 `.lcomm` *symbol* , *length* 40
7.30 `.lflags` 41
7.31 `.line` *line-number* 41
7.32 `.ln` *line-number* 41
7.33 `.list` 41
7.34 `.long` *expressions* 42
7.35 `.nolist` 42
7.36 `.octa` *bignums* 42
7.37 `.org` *new-lc* , *fill* 42
7.38 `.psize` *lines* , *columns* 43
7.39 `.quad` *bignums* 43
7.40 `.sbttl "`*subheading*`"` 43
7.41 `.scl` *class* 43
7.42 `.section` *name*, *subsection* 44
7.43 `.set` *symbol*, *expression* 44
7.44 `.short` *expressions* 44
7.45 `.single` *flonums* 44
7.46 `.size` 44
7.47 `.space` *size* , *fill* 45
7.48 `.stabd, .stabn, .stabs` 45
7.49 `.string "`*str*`"` 46
7.50 `.tag` *structname* 46
7.51 `.text` *subsection* 46
7.52 `.title "`*heading*`"` 46
7.53 `.type` *int* 47
7.54 `.val` *addr* 47
7.55 `.word` *expressions* 47
7.56 Deprecated Directives 48

8 Machine Dependent Features 49

8.1 VAX Dependent Features 49
8.1.1 VAX Command-Line Options 49
8.1.2 VAX Floating Point 50
8.1.3 Vax Machine Directives 50
8.1.4 VAX Opcodes 51
8.1.5 VAX Branch Improvement 51
8.1.6 VAX Operands 53
8.1.7 Not Supported on VAX 53
8.2 AMD 29K Dependent Features 54
8.2.1 Options 54
8.2.2 Syntax 54
8.2.2.1 Special Characters 54
8.2.2.2 Register Names 54
8.2.3 Floating Point 55
8.2.4 AMD 29K Machine Directives 55
8.2.5 Opcodes 55
8.3 H8/300 Dependent Features 56
8.3.1 Options 56
8.3.2 Syntax 56
8.3.2.1 Special Characters 56
8.3.2.2 Register Names 56
8.3.2.3 Addressing Modes 56
8.3.3 Floating Point 57
8.3.4 H8/300 Machine Directives 58
8.3.5 Opcodes 58
8.4 H8/500 Dependent Features 62
8.4.1 Options 62
8.4.2 Syntax 62
8.4.2.1 Special Characters 62
8.4.2.2 Register Names 62
8.4.2.3 Addressing Modes 63
8.4.3 Floating Point 63
8.4.4 H8/500 Machine Directives 63
8.4.5 Opcodes 63
8.5 HPPA Dependent Features 66
8.5.1 Notes 66
8.5.2 Options 66
8.5.3 Syntax 66
8.5.4 Floating Point 67
8.5.5 HPPA Assembler Directives 67
8.5.6 Opcodes 70
8.6 Hitachi SH Dependent Features 71
8.6.1 Options 71
8.6.2 Syntax 71

8.6.2.1 Special Characters 71
8.6.2.2 Register Names 71
8.6.2.3 Addressing Modes 71
8.6.3 Floating Point 72
8.6.4 SH Machine Directives 72
8.6.5 Opcodes 72
8.7 Intel 80960 Dependent Features 75
8.7.1 i960 Command-line Options 75
8.7.2 Floating Point 76
8.7.3 i960 Machine Directives 76
8.7.4 i960 Opcodes 77
8.7.4.1 `callj` 77
8.7.4.2 Compare-and-Branch 78
8.8 M680x0 Dependent Features 79
8.8.1 M680x0 Options 79
8.8.2 Syntax 79
8.8.3 Motorola Syntax 80
8.8.4 Floating Point 81
8.8.5 680x0 Machine Directives 81
8.8.6 Opcodes 82
8.8.6.1 Branch Improvement 82
8.8.6.2 Special Characters 83
8.9 SPARC Dependent Features 84
8.9.1 Options 84
8.9.2 Floating Point 84
8.9.3 Sparc Machine Directives 84
8.10 80386 Dependent Features 86
8.10.1 Options 86
8.10.2 AT&T Syntax versus Intel Syntax 86
8.10.3 Opcode Naming 86
8.10.4 Register Naming 87
8.10.5 Opcode Prefixes 88
8.10.6 Memory References 88
8.10.7 Handling of Jump Instructions 89
8.10.8 Floating Point 90
8.10.9 Writing 16-bit Code 91
8.10.10 Notes 91
8.11 Z8000 Dependent Features 93
8.11.1 Options 93
8.11.2 Syntax 93
8.11.2.1 Special Characters 93
8.11.2.2 Register Names 93
8.11.2.3 Addressing Modes 94
8.11.3 Assembler Directives for the Z8000 94
8.11.4 Opcodes 95

8.12 MIPS Dependent Features 99
8.12.1 Assembler options 99
8.12.2 MIPS ECOFF object code 100
8.12.3 Directives for debugging information 100
8.12.4 Directives to override the ISA level 101

9 Acknowledgements 103

Index 105

1 Overview

This manual is a user guide to the GNU assembler `as`.

Here is a brief summary of how to invoke `as`. For details, see Chapter 2 “Comand-Line Options,” page 9.

```
as [ -a[dhlns] ] [ -D ] [ -f ] [ --help ]
 [ -I dir ] [ -J ] [ -K ] [ -L ] [ -o objfile ]
 [ -R ] [ --statistics ] [ -v ] [ -version ] [ --version ]
 [ -W ] [ -w ] [ -x ] [ -Z ]
 [ -Av6 | -Av7 | -Av8 | -Av9 | -Asparclite | -bump ]
 [ -ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC ]
 [ -b ] [ -no-relax ]
 [ -l ] [ -m68000 | -m68010 | -m68020 | ... ]
 [ -nocpp ] [ -EL ] [ -EB ] [ -G num ] [ -mcpu=CPU ]
 [ -mips1 ] [ -mips2 ] [ -mips3 ] [ -m4650 ] [ -no-m4650 ]
 [ --trap ] [ --break ]
 [ -- | files ... ]
```

`-a[dhlns]`
: Turn on listings, in any of a variety of ways:

  `-ad` omit debugging directives

  `-ah` include high-level source

  `-al` include assembly

  `-an` omit forms processing

  `-as` include symbols

  You may combine these options; for example, use ‘`-aln`’ for assembly listing without forms processing. By itself, ‘`-a`’ defaults to ‘`-ahls`’—that is, all listings turned on.

`-D`
: Ignored. This option is accepted for script compatibility with calls to other assemblers.

`-f`
: “fast”—skip whitespace and comment preprocessing (assume source is compiler output).

`--help`
: Print a summary of the command line options and exit.

`-I` *dir*
: Add directory *dir* to the search list for `.include` directives.

`-J`
: Don’t warn about signed overflow.

`-K`
: Issue warnings when difference tables altered for long displacements.

`-L`
: Keep (in the symbol table) local symbols, starting with ‘`L`’.

`-o` *objfile*
: Name the object-file output from `as` *objfile*.

`-R` Fold the data section into the text section.

`--statistics`
: Print the maximum space (in bytes) and total time (in seconds) used by assembly.

`-v`
`-version` Print the `as` version.

`--version`
: Print the `as` version and exit.

`-W` Suppress warning messages.

`-w` Ignored.

`-x` Ignored.

`-Z` Generate an object file even after errors.

`-- | `*`files`*` . . .`
: Standard input, or source files to assemble.

The following options are available when as is configured for the Intel 80960 processor.

`-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC`
: Specify which variant of the 960 architecture is the target.

`-b` Add code to collect statistics about branches taken.

`-no-relax`
: Do not alter compare-and-branch instructions for long displacements; error if necessary.

The following options are available when as is configured for the Motorola 68000 series.

`-l` Shorten references to undefined symbols, to one word instead of two.

`-m68000 | -m68008 | -m68010 | -m68020 | -m68030 | -m68040 | -m68302 | -m68331 | -m68332 | -m68333 | -m68340 | -mcpu32`
: Specify what processor in the 68000 family is the target. The default is normally the 68020, but this can be changed at configuration time.

`-m68881 | -m68882 | -mno-68881 | -mno-68882`
: The target machine does (or does not) have a floating-point coprocessor. The default is to assume a coprocessor for 68020, 68030, and cpu32. Although the basic 68000 is not compatible with the 68881, a combination of the two can be specified, since it’s possible to do emulation of the coprocessor instructions with the main processor.

`-m68851 | -mno-68851`
: The target machine does (or does not) have a memory-management unit coprocessor. The default is to assume an MMU for 68020 and up.

The following options are available when `as` is configured for the SPARC architecture:

`-Av6 | -Av7 | -Av8 | -Av9 | -Asparclite`
: Explicitly select a variant of the SPARC architecture.

`-bump` Warn when the assembler switches to another architecture.

The following options are available when as is configured for a MIPS processor.

`-G` *num* This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format, such as a DECstation running Ultrix. The default value is 8.

`-EB` Generate "big endian" format output.

`-EL` Generate "little endian" format output.

`-mips1`
`-mips2`
`-mips3` Generate code for a particular MIPS Instruction Set Architecture level. '`-mips1`' corresponds to the R2000 and R3000 processors, '`-mips2`' to the R6000 processor, and '`-mips3`' to the R4000 processor.

`-m4650`

`-no-m4650`
: Generate code for the MIPS R4650 chip. This tells the assembler to accept the '`mad`' and '`madu`' instruction, and to not schedule '`nop`' instructions around accesses to the '`HI`' and '`LO`' registers. '`-no-m4650`' turns off this option.

`-mcpu=`*CPU*
: Generate code for a particular MIPS cpu. This has little effect on the assembler, but it is passed by `gcc`.

`-nocpp` `as` ignores this option. It is accepted for compatibility with the native tools.

`--trap`
`--no-trap`
`--break`
`--no-break`
: Control how to deal with multiplication overflow and division by zero. '`--trap`' or '`--no-break`' (which are synonyms)

take a trap exception (and only work for Instruction Set Architecture level 2 and higher); '`--break`' or '`--no-trap`' (also synonyms, and the default) take a break exception.

1.1 Structure of this Manual

This manual is intended to describe what you need to know to use GNU `as`. We cover the syntax expected in source files, including notation for symbols, constants, and expressions; the directives that `as` understands; and of course how to invoke `as`.

This manual also describes some of the machine-dependent features of various flavors of the assembler.

On the other hand, this manual is *not* intended as an introduction to programming in assembly language—let alone programming in general! In a similar vein, we make no attempt to introduce the machine architecture; we do *not* describe the instruction set, standard mnemonics, registers or addressing modes that are standard to a particular architecture. You may want to consult the manufacturer's machine architecture manual for this information.

1.2 as, the GNU Assembler

GNU `as` is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax.

`as` is primarily intended to assemble the output of the GNU C compiler `gcc` for use by the linker `ld`. Nevertheless, we've tried to make `as` assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (see Chapter 8 "Machine Dependencies," page 49). This doesn't mean `as` always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Unlike older assemblers, `as` is designed to assemble a source program in one pass of the source file. This has a subtle impact on the `.org` directive (see Section 7.37 "`.org`," page 42).

1.3 Object File Formats

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See Section 5.5 “Symbol Attributes,” page 29.

1.4 Command Line

After the program name `as`, the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

‘`--`’ (two hyphens) by itself names the standard input file explicitly, as one of the files for `as` to assemble.

Except for ‘`--`’ any command line argument that begins with a hyphen (‘`-`’) is an option. Each option changes the behavior of `as`. No option changes the way another option works. An option is a ‘`-`’ followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option’s letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
as -o my-object-file.o mumble.s
as -omy-object-file.o mumble.s
```

1.5 Input Files

We use the phrase *source program*, abbreviated *source*, to describe the program input to one run of `as`. The program may be in one or more files; how the source is partitioned into files doesn’t change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run `as` it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give `as` a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give `as` no file names it attempts to read one input file from the `as` standard input, which is normally your terminal. You may have to type `CTL-D` to tell `as` there is no more program to assemble.

Use '`--`' if you need to explicitly name the standard input file in your command line.

If the source is empty, `as` produces a small, empty object file.

Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See Section 1.7 "Error and Warning Messages," page 6.

Physical files are those files named in the command line given to `as`.

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when `as` source is itself synthesized from other files. See Section 7.4 "`.app-file`," page 36.

1.6 Output (Object) File

Every time you run `as` it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is `a.out`, or `b.out` when `as` is configured for the Intel 80960. You can give it another name by using the `-o` option. Conventionally, object file names end with '`.o`'. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn't currently possible, but it can be done for the `a.out` format.)

The object file is meant for input to the linker `ld`. It contains assembled program code, information to help `ld` integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

1.7 Error and Warning Messages

`as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs `as` automatically. Warnings report an assumption made so that `as` could keep assembling a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where **NNN** is a line number). If a logical file name has been given (see Section 7.4 "`.app-file`," page 36) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see Section 7.31 "`.line`," page 41) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the grand Unix tradition).

Error messages have the format

```
file_name:NNN:FATAL:Error Message Text
```

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

2 Command-Line Options

This chapter describes command-line options available in *all* versions of the GNU assembler; see Chapter 8 “Machine Dependencies,” page 49, for options specific to particular machine architectures.

If you are invoking `as` via the GNU C compiler (version 2), you can use the ‘`-Wa`’ option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the ‘`-Wa`’) by commas. For example:

```
gcc -c -g -O -Wa,-alh,-L file.c
```

emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this ‘`-Wa`’ mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the ‘`-v`’ option to see precisely what options it passes to each compilation pass, including the assembler.)

2.1 Enable Listings: `-a[dhlns]`

These options enable listing output from the assembler. By itself, ‘`-a`’ requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: ‘`-ah`’ requests a high-level language listing, ‘`-al`’ requests an output-program assembly listing, and ‘`-as`’ requests a symbol table listing. High-level listings require that a compiler debugging option like ‘`-g`’ be used, and that assembly listings (‘`-al`’) be requested also.

Use the ‘`-ad`’ option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list`, `.nolist`, `.psize`, `.eject`, `.title`, and `.sbttl`. The ‘`-an`’ option turns off all forms processing. If you do not request listing output with one of the ‘`-a`’ options, the listing-control directives have no effect.

The letters after ‘`-a`’ may be combined into one option, *e.g.*, ‘`-aln`’.

2.2 `-D`

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with `as`.

2.3 Work Faster: `-f`

'`-f`' should only be used when assembling programs written by a (trusted) compiler. '`-f`' stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. See Section 3.1 "Preprocessing," page 13.

> *Warning:* if you use '`-f`' when the files actually need to be preprocessed (if they contain comments, for example), `as` does not work correctly.

2.4 `.include` search path: `-I` *`path`*

Use this option to add a *`path`* to the list of directories `as` searches for files specified in `.include` directives (see Section 7.27 "`.include`," page 40). You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, `as` searches any '`-I`' directories in the same order as they were specified (left to right) on the command line.

2.5 Difference Tables: `-K`

`as` sometimes alters the code emitted for directives of the form '`.word` *`sym1-sym2`*'; see Section 7.55 "`.word`," page 47. You can use the '`-K`' option if you want a warning issued when this is done.

2.6 Include Local Labels: `-L`

Labels beginning with '`L`' (upper case only) are called *local labels*. See Section 5.3 "Symbol Names," page 27. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `as` and `ld` discard such labels, so you do not normally debug with them.

This option tells `as` to retain those '`L. . .`' symbols in the object file. Usually if you do this you also tell the linker `ld` to preserve symbols whose names begin with '`L`'.

By default, a local label is any label beginning with '`L`', but each target is allowed to redefine the local label prefix. On the HPPA local labels begin with '`L$`'.

2.7 Name the Object File: -o

There is always one object file output when you run as. By default it has the name 'a.out' (or 'b.out', for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, as overwrites any existing file of the same name.

2.8 Join Data and Text Sections: -R

-R tells as to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See Chapter 4 "Sections and Relocation," page 21.)

When you specify -R it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of as. In future, -R may work this way.

When as is configured for COFF output, this option is only useful if you use sections named '.text' and '.data'.

-R is not supported for any of the HPPA targets. Using -R generates a warning from as.

2.9 Display Assembly Statistics: --statistics

Use '--statistics' to display two statistics about the resources used by as: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

2.10 Announce Version: -v

You can find out what version of as is running by including the option '-v' (which you can also spell as '-version') on the command line.

2.11 Suppress Warnings: -W

as should never give a warning or error message when assembling compiler output. But programs written by people often cause as to give a

warning that a particular assumption was made. All such warnings are directed to the standard error file. If you use this option, no warnings are issued. This option only affects the warning messages: it does not change any particular of how `as` assembles your file. Errors, which stop the assembly, are still reported.

2.12 Generate Object File in Spite of Errors: `-Z`

After an error message, `as` normally produces no output. If for some reason you are interested in object file output even after `as` gives an error message on your program, use the '`-Z`' option. If there are any errors, `as` continues anyways, and writes an object file after a final warning message of the form '*n* `errors,` *m* `warnings, generating bad object file.`'

3 Syntax

This chapter describes the machine-independent syntax allowed in a source file. `as` syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler, except that `as` does not assemble Vax bit-fields.

3.1 Preprocessing

The `as` internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (see Section 7.27 "`.include`," page 40). You can use the GNU C compiler driver to get other "CPP" style preprocessing, by giving the input file a '`.S`' suffix. See section "Options Controlling the Kind of Output" in *Using GNU CC*.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is `#NO_APP` or if you use the '`-f`' option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the by putting a line that says `#APP` before the text that may contain whitespace or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intend to support `asm` statements in compilers whose output is otherwise free of comments and whitespace.

3.2 Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see Section 3.6.1 "Character Constants," page 16), any whitespace means the same as exactly one space.

3.3 Comments

There are two ways of rendering comments to `as`. In both cases the comment is equivalent to one space.

Anything from '/*' through the next '*/' is a comment. This means you may not nest these comments.

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/

/* This sort of comment does not nest. */
```

Anything from the *line comment* character to the next newline is considered a comment and is ignored. The line comment character is '#' on the Vax; '#' on the i960; '!' on the SPARC; '|' on the 680x0; ';' for the AMD 29K family; ';' for the H8/300 family; '!' for the H8/500 family; ';' for the HPPA; '!' for the Hitachi SH; '!' for the Z8000; see Chapter 8 "Machine Dependencies," page 49.

On some machines there are two different line comment characters. One character only begins a comment if it is the first non-whitespace character on a line, while the other always begins a comment.

To be compatible with past assemblers, lines that begin with '#' have a special interpretation. Following the '#' should be an absolute expression (see Chapter 6 "Expressions," page 31): the logical line number of the *next* line. Then a string (see Section 3.6.1.1 "Strings," page 16) is allowed: if present it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                          # This is an ordinary comment.
# 42-6 "new_file_name"    # New logical file name
                          # This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of `as`.

3.4 Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters '_.$'. On most machines, you can also use `$` in symbol names; exceptions are noted in Chapter 8 "Machine Dependencies," page 49. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end

with a newline, the end of a file is not a possible symbol delimiter). See Chapter 5 “Symbols,” page 27.

3.5 Statements

A *statement* ends at a newline character (‘\n’) or line separator character. (The line separator is usually ‘;’, unless this conflicts with the comment character; see Chapter 8 “Machine Dependencies,” page 49.) The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When `as` reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot ‘.’ then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language *instruction*: it assembles into a machine language instruction. Different versions of `as` for different computers recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer’s assembly language.

A label is a symbol immediately followed by a colon (`:`). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label’s symbol and its colon. See Section 5.1 “Labels,” page 27.

For HPPA targets, labels need not be immediately followed by a colon, but the definition of a label must begin in column zero. This also implies that only one label may be defined on each line.

```
label:     .directive    followed by something
another_label:           # This is an empty statement.
           instruction   operand_1, operand_2, ...
```

3.6 Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                 # - pi, a flonum.
```

3.6.1 Character Constants

There are two kinds of character constants. A *character* stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string *literals*) are potentially many bytes and their values may not be used in arithmetic expressions.

3.6.1.1 Strings

A *string* is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to *escape* these characters: precede them with a backslash '\' character. For example '\\' represents one backslash: the first \ is an escape which tells `as` to interpret the second character literally as a backslash (which prevents `as` from recognizing the second \ as an escape character). The complete list of escapes follows.

`\b` Mnemonic for backspace; for ASCII this is octal code 010.

`\f` Mnemonic for FormFeed; for ASCII this is octal code 014.

`\n` Mnemonic for newline; for ASCII this is octal code 012.

`\r` Mnemonic for carriage-Return; for ASCII this is octal code 015.

`\t` Mnemonic for horizontal Tab; for ASCII this is octal code 011.

`\` *digit digit digit*
: An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, `\008` has the value 010, and `\009` the value 011.

`\x` *hex-digit hex-digit*
: A hex character code. The numeric code is 2 hexadecimal digits. Either upper or lower case `x` works.

`\\` Represents one '\' character.

`\"` Represents one '"' character. Needed in strings to represent this character, because an unescaped '"' would end the string.

`\` *anything-else*
Any other character when escaped by \ gives a warning, but assembles as if the '\' was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However `as` has no other interpretation, so `as` knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

3.6.1.2 Characters

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write `'\\` where the first \ escapes the second \. As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. `as` assumes your character code is ASCII: `'A` means 65, `'B` means 66, and so on.

3.6.2 Number Constants

`as` distinguishes three kinds of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an `int` in the C language. *Bignums* are integers, but they are stored in more than 32 bits. *Flonums* are floating point numbers, described below.

3.6.2.1 Integers

A binary integer is '`0b`' or '`0B`' followed by zero or more of the binary digits '`01`'.

An octal integer is '`0`' followed by zero or more of the octal digits ('`01234567`').

A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789`).

A hexadecimal integer is `0x` or `0X` followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF`.

Integers have the usual values. To denote a negative integer, use the prefix operator `-` discussed under expressions (see Section 6.2.3 “Prefix Operators,” page 32).

3.6.2.2 Bignums

A *bignum* has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

3.6.2.3 Flonums

A *flonum* represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by `as` to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer’s floating point format (or formats) by a portion of `as` specialized to that computer.

A flonum is written by writing (in order)

- The digit `0`. (`0` is optional on the HPPA.)
- A letter, to tell `as` the rest of the number is a flonum. `e` is recommended. Case is not important.

  On the H8/300, H8/500, Hitachi SH, and AMD 29K architectures, the letter must be one of the letters `DFPRSX` (in upper or lower case).

  On the Intel 960 architecture, the letter must be one of the letters `DFT` (in upper or lower case).

  On the HPPA architecture, the letter must be `E` (upper case only).
- An optional sign: either `+` or `-`.
- An optional *integer part*: zero or more decimal digits.
- An optional *fractional part*: `.` followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An `E` or `e`.
  - Optional sign: either `+` or `-`.
  - One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

`as` does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running `as`.

4 Sections and Relocation

4.1 Background

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker `ld` reads many object files (partial programs) and combines their contents to form a runnable program. When `as` emits an object file, the partial program is assumed to start at address 0. `ld` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how `as` uses sections.

`ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. For the H8/300 and H8/500, and for the Hitachi SH, `as` pads sections if needed to ensure they end on a word (sixteen bit) boundary.

An object file written by `as` has at least three sections, any of which may be empty. These are named *text*, *data* and *bss* sections.

When it generates COFF output, `as` can also generate whatever other named sections you specify using the '`.section`' directive (see Section 7.42 "`.section`," page 44). If you do not use any directives that place output in the '`.text`' or '`.data`' sections, these sections still exist, but are empty.

When `as` generates SOM or ELF output for the HPPA, `as` can also generate whatever other named sections you specify using the '`.space`' and '`.subspace`' directives. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for details on the '`.space`' and '`.subspace`' assembler directives.

Additionally, `as` uses different names for the standard text, data, and bss sections when generating SOM output. Program text is placed into the '`$CODE$`' section, data into '`$DATA$`', and BSS into '`$BSS$`'.

Within the object file, the text section starts at address `0`, the data section follows, and the bss section follows the data section.

When generating either SOM or ELF output files on the HPPA, the text section starts at address `0`, the data section at address `0x4000000`, and the bss section follows the data section.

To let `ld` know which data changes when the sections are relocated, and how to change that data, `as` also writes to the object file details of the relocation needed. To perform relocation `ld` must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of

  (*address*) − (*start-address of section*)?
- Is the reference to an address "Program-Counter relative"?

In fact, every address `as` ever uses is expressed as

(*section*) + (*offset into section*)

Further, most expressions `as` computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.)

In this manual we use the notation {*secname N*} to mean "offset *N* into section *secname*."

Apart from text, data and bss sections you need to know about the *absolute* section. When `ld` mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is "relocated" to run-time address 0 by `ld`. Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the program.

The idea of sections is extended to the *undefined* section. Any address whose section is unknown at assembly time is by definition rendered {undefined *U*}—where *U* is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy the word *section* is used to describe groups of sections in the linked program. `ld` puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

Some sections are manipulated by `ld`; others are invented for use of `as` and have no meaning except during assembly.

4.2 ld Sections

`ld` deals with just four kinds of sections, summarized below.

named sections
text section
data section

These sections hold your program. `as` and `ld` treat them as separate but equal sections. Anything you can say of one section is true another. When the program is running, however, it is customary for the text section to be unalterable. The text section is often shared among processes: it contains instructions, constants and the like. The data section of a running program is usually alterable: for example, C variables would be stored in the data section.

bss section

This section contains zeroed bytes when your program begins running. It is used to hold unitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

absolute section

Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that `ld` must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

undefined section

This "section" is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. The example uses the traditional section names '`.text`' and '`.data`'. Memory addresses are on the horizontal axis.

Partial program #1:

```
text          data       bss
 ttttt         dddd       00
```

Partial program #2:

```
text  data     bss
 TTT   DDDD     000
```

linked program:

```
    text                          data                bss
   TTT   ttttt             dddd    DDDD    00000    ...
```

addresses:

0...

4.3 as Internal Sections

These sections are meant only for the internal use of `as`. They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in `as` warning messages, so it might be helpful to have an idea of their meanings to `as`. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

ASSEMBLER-INTERNAL-LOGIC-ERROR!
: An internal assembler logic error has been found. This means there is a bug in the assembler.

expr section
: The assembler stores complex expression internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the expr section.

4.4 Sub-Sections

Assembled bytes conventionally fall into two sections: text and data. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. `as` allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a '`.text 0`' before

each section of code being output, and a '`.text 1`' before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes. (Subsections may be padded a different amount on different flavors of `as`.)

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; `ld` and other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a '`.text` *expression*' or a '`.data` *expression*' statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: '`.section` *name*, *expression*'. *Expression* should be an absolute expression. (See Chapter 6 "Expressions," page 31.) If you just say '`.text`' then '`.text 0`' is assumed. Likewise '`.data`' means '`.data 0`'. Assembly begins in `text 0`. For instance:

```
.text 0     # The default subsection is text 0 anyway.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

Each section has a *location counter* incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to `as` there is no concept of a subsection location counter. There is no way to directly manipulate a location counter—but the `.align` directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the *active* location counter.

4.5 bss Section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

Addresses in the bss section are allocated with special directives; you may not assemble anything directly into the bss section. Hence there are no bss subsections. See Section 7.8 "`.comm`," page 36, see Section 7.29 "`.lcomm`," page 40.

5 Symbols

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

> *Warning:* `as` does not place symbols in the object file in the same order they were declared. This may break some debuggers.

5.1 Labels

A *label* is written as a symbol immediately followed by a colon '`:`'. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

On the HPPA, the usual form for a label need not be immediately followed by a colon, but instead must start in column zero. Only one label may be defined on a single line. To work around this, the HPPA version of `as` also provides a special directive `.label` for defining labels more flexibly.

5.2 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign '`=`', followed by an expression (see Chapter 6 "Expressions," page 31). This is equivalent to using the `.set` directive. See Section 7.43 "`.set`," page 44.

5.3 Symbol Names

Symbol names begin with a letter or with one of '`._`'. On most machines, you can also use `$` in symbol names; exceptions are noted in Chapter 8 "Machine Dependencies," page 49. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 8 "Machine Dependencies," page 49), and underscores. For the AMD 29K family, '`?`' is also allowed in the body of a symbol name, though not at its beginning.

Case of letters is significant: `foo` is a different symbol name than `Foo`.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names '0' '1' ... '9'. To define a local symbol, write a label of the form '**N**:' (where **N** represents any digit). To refer to the most recent previous definition of that symbol write '**N**b', using the same digit as when you defined the label. To refer to the next definition of a local label, write '**N**f'—where **N** gives you a choice of 10 forward references. The 'b' stands for "backwards" and the 'f' stands for "forwards".

Local symbols are not emitted by the current GNU C compiler.

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

`L`
: All local labels begin with 'L'. Normally both `as` and `ld` forget symbols that start with 'L'. These labels are used for symbols you are never intended to see. If you use the '-L' option then `as` retains these symbols in the object file. If you also instruct `ld` to retain these symbols, you may use them in debugging.

digit
: If the label is written '0:' then the digit is '0'. If the label is written '1:' then the digit is '1'. And so on up through '9:'.

`^A`
: This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value '\001'.

ordinal number
: This is a serial number to keep the labels distinct. The first '0:' gets the number '1'; The 15th '0:' gets the number '15'; *etc.*. Likewise for the other labels '1:' through '9:'.

For instance, the first `1:` is named `L1^A1`, the 44th `3:` is named `L3^A44`.

5.4 The Special Dot Symbol

The special symbol '.' refers to the current address that `as` is assembling into. Thus, the expression '`melvin: .long .`' defines `melvin` to contain its own address. Assigning a value to `.` is treated the same

as a `.org` directive. Thus, the expression '`.=.+4`' is the same as saying '`.space 4`'.

5.5 Symbol Attributes

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, `as` assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

5.5.1 Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as `ld` changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and `ld` tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a `.comm` common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

5.5.2 Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

5.5.3 Symbol Attributes: `a.out`

5.5.3.1 Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (see Section 7.11 "`.desc`," page 37). A descriptor value means nothing to `as`.

5.5.3.2 Other

This is an arbitrary 8-bit value. It means nothing to `as`.

5.5.4 Symbol Attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between `.def` and `.endef` directives.

5.5.4.1 Primary Attributes

The symbol name is set with `.def`; the value and type, respectively, with `.val` and `.type`.

5.5.4.2 Auxiliary Attributes

The `as` directives `.dim`, `.line`, `.scl`, `.size`, and `.tag` can generate auxiliary symbol table information for COFF.

5.5.5 Symbol Attributes for SOM

The SOM format for the HPPA supports a multitude of symbol attributes set with the `.EXPORT` and `.IMPORT` directives.

The attributes are described in *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) under the `IMPORT` and `EXPORT` assembler directive documentation.

6 Expressions

An *expression* specifies an address or numeric value. Whitespace may precede and/or follow an expression.

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when `as` sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression—but the second pass is currently not implemented. `as` aborts with an error message in this situation.

6.1 Empty Expressions

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression, and `as` assumes a value of (absolute) 0. This is compatible with other assemblers.

6.2 Integer Expressions

An *integer expression* is one or more *arguments* delimited by *operators*.

6.2.1 Arguments

Arguments are symbols, numbers or subexpressions. In other contexts arguments are sometimes called "arithmetic operands". In this manual, to avoid confusing them with the "instruction operands" of the machine language, we use the term "argument" to refer to parts of expressions only, reserving the word "operand" to refer only to machine instruction operands.

Symbols are evaluated to yield {*section NNN*} where *section* is one of text, data, bss, absolute, or undefined. *NNN* is a signed, 2's complement 32 bit integer.

Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and `as` pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis '`(`' followed by an integer expression, followed by a right parenthesis '`)`'; or a prefix operator followed by an argument.

6.2.2 Operators

Operators are arithmetic functions, like `+` or `%`. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by whitespace.

6.2.3 Prefix Operator

`as` has the following *prefix operators*. They each take one argument, which must be absolute.

`-` *Negation*. Two's complement negation.

`~` *Complementation*. Bitwise not.

6.2.4 Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from `+` or `-`, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

   `*` *Multiplication*.

   `/` *Division*. Truncation is the same as the C operator '`/`'

   `%` *Remainder*.

   `<`
   `<<` *Shift Left*. Same as the C operator '`<<`'.

   `>`
   `>>` *Shift Right*. Same as the C operator '`>>`'.

2. Intermediate precedence

   `|`
   Bitwise Inclusive Or.

   `&` *Bitwise And*.

   `^` *Bitwise Exclusive Or*.

   `!` *Bitwise Or Not*.

3. Lowest Precedence

   `+` *Addition*. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

– *Subtraction*. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

7 Assembler Directives

All assembler directives have names that begin with a period ('.'). The rest of the name is letters, usually in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 8 "Machine Dependencies," page 49.

7.1 `.abort`

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells `as` to quit also. One day `.abort` will not be supported.

7.2 `.ABORT`

When producing COFF output, `as` accepts this directive as a synonym for '`.abort`'.

When producing `b.out` output, `as` accepts this directive, but ignores it.

7.3 `.align` *abs-expr* , *abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example '`.align 3`' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

For the HPPA, the first expression (which must be absolute) is the alignment request in bytes. For example '`.align 8`' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.

7.4 .app-file *string*

`.app-file` (which may also be spelled '`.file`') tells `as` that we are about to start a new logical file. *string* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '`"`'; but if you wish to specify an empty file name is permitted, you must give the quotes–`""`. This statement may go away in future: it is only recognized to be compatible with old `as` programs.

7.5 .ascii "*string*"...

`.ascii` expects zero or more string literals (see Section 3.6.1.1 "Strings," page 16) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.6 .asciz "*string*"...

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The "z" in '`.asciz`' stands for "zero".

7.7 .byte *expressions*

`.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

7.8 .comm *symbol* , *length*

`.comm` declares a named common area in the bss section. Normally `ld` reserves memory addresses for it during linking, so no partial program defines the location of the symbol. Use `.comm` to tell `ld` that it must be at least *length* bytes long. `ld` allocates space for each `.comm` symbol that is at least as long as the longest `.comm` request in any of the partial programs linked. *length* is an absolute expression.

The syntax for `.comm` differs slightly on the HPPA. The syntax is '*symbol* `.comm,` *length*'; *symbol* is optional.

7.9 .data *subsection*

`.data` tells `as` to assemble the following statements onto the end of the data subsection numbered *subsection* (which is an absolute expression). If *subsection* is omitted, it defaults to zero.

7.10 `.def name`

Begin defining debugging information for a symbol *name*; the definition extends until the `.endef` directive is encountered.

This directive is only observed when `as` is configured for COFF format output; when producing `b.out`, '`.def`' is recognized, but ignored.

7.11 `.desc symbol, abs-expression`

This directive sets the descriptor of the symbol (see Section 5.5 "Symbol Attributes," page 29) to the low 16 bits of an absolute expression.

The '`.desc`' directive is not available when `as` is configured for COFF output; it is only for `a.out` or `b.out` object format. For the sake of compatibility, `as` accepts it, but produces no output, when configured for COFF.

7.12 `.dim`

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs.

'`.dim`' is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

7.13 `.double flonums`

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 "Machine Dependencies," page 49.

7.14 `.eject`

Force a page break at this point, when generating assembly listings.

7.15 `.else`

`.else` is part of the `as` support for conditional assembly; see Section 7.26 "`.if`," page 40. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.16 .endef

This directive flags the end of a symbol definition begun with `.def`.

'`.endef`' is only meaningful when generating COFF format output; if `as` is configured to generate `b.out`, it accepts this directive but ignores it.

7.17 .endif

`.endif` is part of the `as` support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See Section 7.26 "`.if`," page 40.

7.18 .equ *symbol, expression*

This directive sets the value of *symbol* to *expression*. It is synonymous with '`.set`'; see Section 7.43 "`.set`," page 44.

The syntax for `equ` on the HPPA is '*symbol* `.equ` *expression*'.

7.19 .extern

`.extern` is accepted in the source program—for compatibility with other assemblers—but it is ignored. `as` treats all undefined symbols as external.

7.20 .file *string*

`.file` (which may also be spelled '`.app-file`') tells `as` that we are about to start a new logical file. *string* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '`"`'; but if you wish to specify an empty file name, you must give the quotes–`""`. This statement may go away in future: it is only recognized to be compatible with old `as` programs. In some configurations of `as`, `.file` has already been removed to avoid conflicts with other assemblers. See Chapter 8 "Machine Dependencies," page 49.

7.21 .fill *repeat* , *size* , *value*

result, *size* and *value* are absolute expressions. This emits *repeat* copies of *size* bytes. *Repeat* may be zero or more. *Size* may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each *repeat*

bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are *value* rendered in the byte-order of an integer on the computer `as` is assembling for. Each *size* bytes in a repetition is taken from the lowest order *size* bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and *value* are optional. If the second comma and *value* are absent, *value* is assumed zero. If the first comma and following tokens are absent, *size* is assumed to be 1.

7.22 `.float` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.single`. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 “Machine Dependencies,” page 49.

7.23 `.global` *symbol*, `.globl` *symbol*

`.global` makes the symbol visible to `ld`. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings (‘`.globl`’ and ‘`.global`’) are accepted, for compatibility with other assemblers.

On the HPPA, `.global` is not always enough to make it accessible to other partial programs. You may need the HPPA-only `.EXPORT` directive as well. See Section 8.5.5 “HPPA Assembler Directives,” page 67.

7.24 `.hword` *expressions*

This expects zero or more *expressions*, and emits a 16 bit number for each.

This directive is a synonym for ‘`.short`’; depending on the target architecture, it may also be a synonym for ‘`.word`’.

7.25 `.ident`

This directive is used by some assemblers to place tags in object files. `as` simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

7.26 `.if` *absolute expression*

`.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an *absolute expression*) is non-zero. The end of the conditional section of code must be marked by `.endif` (see Section 7.17 "`.endif`," page 38); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.15 "`.else`," page 37.

The following variants of `.if` are also supported:

`.ifdef` *symbol*
: Assembles the following section of code if the specified *symbol* has been defined.

`.ifndef` *symbol*
`ifnotdef` *symbol*
: Assembles the following section of code if the specified *symbol* has not been defined. Both spelling variants are equivalent.

7.27 `.include "`*file*`"`

This directive provides a way to include supporting files at specified points in your source program. The code from *file* is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the '`-I`' command-line option (see Chapter 2 "Command-Line Options," page 9). Quotation marks are required around *file*.

7.28 `.int` *expressions*

Expect zero or more *expressions*, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

7.29 `.lcomm` *symbol* , *length*

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *Symbol* is not declared global (see Section 7.23 "`.global`," page 39), so is normally not visible to `ld`.

The syntax for `.lcomm` differs slightly on the HPPA. The syntax is '*symbol* `.lcomm,` *length*'; *symbol* is optional.

7.30 `.lflags`

`as` accepts this directive, for compatibility with other assemblers, but ignores it.

7.31 `.line` *line-number*

Change the logical line number. *line-number* must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number *line-number* − 1. One day `as` will no longer support this directive: it is recognized only for compatibility with existing assembler programs.

Warning: In the AMD29K configuration of as, this command is not available; use the synonym `.ln` in that context.

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, `as` still recognizes it when producing COFF output, and treats '`.line`' as though it were the COFF '`.ln`' *if* it is found outside a `.def`/`.endef` pair.

Inside a `.def`, '`.line`' is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

7.32 `.ln` *line-number*

'`.ln`' is a synonym for '`.line`'.

7.33 `.list`

Control (in conjunction with the `.nolist` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the '`-a`' command line option; see Chapter 2 "Command-Line Options," page 9), the initial value of the listing counter is one.

7.34 `.long` *`expressions`*

`.long` is the same as '`.int`', see Section 7.28 "`.int`," page 40.

7.35 `.nolist`

Control (in conjunction with the `.list` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

7.36 `.octa` *`bignums`*

This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer.

The term "octa" comes from contexts in which a "word" is two bytes; hence *octa*-word for 16 bytes.

7.37 `.org` *`new-lc , fill`*

Advance the location counter of the current section to *`new-lc`*. *`new-lc`* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use `.org` to cross sections: if *`new-lc`* has the wrong section, the `.org` directive is ignored. To be compatible with former assemblers, if the section of *`new-lc`* is absolute, `as` issues a warning, then pretends the section of *`new-lc`* is the same as the current subsection.

`.org` may only increase the location counter, or leave it unchanged; you cannot use `.org` to move the location counter backwards.

Because `as` tries to assemble programs in one pass, *`new-lc`* may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *`fill`* which should be an absolute expression. If the comma and *`fill`* are omitted, *`fill`* defaults to zero.

7.38 `.psize` *`lines , columns`*

Use this directive to declare the number of lines—and, optionally, the number of columns—to use for each page, when generating listings.

If you do not use `.psize`, listings use a default line-count of 60. You may omit the comma and *`columns`* specification; the default width is 200 columns.

`as` generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify *`lines`* as `0`, no formfeeds are generated save those explicitly specified with `.eject`.

7.39 `.quad` *`bignums`*

`.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum.

The term "quad" comes from contexts in which a "word" is two bytes; hence *quad*-word for 8 bytes.

7.40 `.sbttl "`*`subheading`*`"`

Use *`subheading`* as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

7.41 `.scl` *`class`*

Set the storage-class value for a symbol. This directive may only be used inside a `.def`/`.endef` pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

The '`.scl`' directive is primarily associated with COFF output; when configured to generate `b.out` output format, `as` accepts this directive but ignores it.

7.42 `.section` *name, subsection*

Assemble the following code into end of subsection numbered *subsection* in the COFF named section *name*. If you omit *subsection*, `as` uses subsection number zero. '`.section .text`' is equivalent to the `.text` directive; '`.section .data`' is equivalent to the `.data` directive. This directive is only supported for targets that actually support arbitrarily named sections; on `a.out` targets, for example, it is not accepted, even with a standard `a.out` section name as its parameter.

7.43 `.set` *symbol, expression*

Set the value of *symbol* to *expression*. This changes *symbol*'s value and type to conform to *expression*. If *symbol* was flagged as external, it remains flagged. (See Section 5.5 "Symbol Attributes," page 29.)

You may `.set` a symbol many times in the same assembly.

If you `.set` a global symbol, the value stored in the object file is the last value stored into it.

The syntax for `set` on the HPPA is '*symbol* `.set` *expression*'.

7.44 `.short` *expressions*

`.short` is normally the same as '`.word`'. See Section 7.55 "`.word`," page 47.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; see Chapter 8 "Machine Dependencies," page 49.

7.45 `.single` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.float`. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 "Machine Dependencies," page 49.

7.46 `.size`

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs.

'`.size`' is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

7.47 `.space size , fill`

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

> *Warning:* `.space` has a completely different meaning for HPPA targets; use `.block` as a substitute. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for the meaning of the `.space` directive. See Section 8.5.5 “HPPA Assembler Directives,” page 67, for a summary.

On the AMD 29K, this directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

> *Warning:* In most versions of the GNU assembler, the directive `.space` has the effect of `.block` See Chapter 8 “Machine Dependencies,” page 49.

7.48 `.stabd, .stabn, .stabs`

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 “Symbols,” page 27), for use by symbolic debuggers. The symbols are not entered in the `as` hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

string This is the symbol's name. It may contain any character except '`\000`', so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.

type An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but `ld` and debuggers choke on silly bit patterns.

other An absolute expression. The symbol's “other” attribute is set to the low 8 bits of this expression.

desc An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

value An absolute expression which becomes the symbol's value.

If a warning is detected while reading a `.stabd`, `.stabn`, or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

`.stabd` *type* , *other* , *desc*

The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

`.stabn` *type* , *other* , *desc* , *value*

The name of the symbol is set to the empty string `""`.

`.stabs` *string* , *type* , *other* , *desc* , *value*

All five fields are specified.

7.49 .string "*str*"

Copy the characters in *str* to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in Section 3.6.1.1 "Strings," page 16.

7.50 .tag *structname*

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. Tags are used to link structure definitions in the symbol table with instances of those structures.

'`.tag`' is only used when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

7.51 .text *subsection*

Tells `as` to assemble the following statements onto the end of the text subsection numbered *subsection*, which is an absolute expression. If *subsection* is omitted, subsection number zero is used.

7.52 .title "*heading*"

Use *heading* as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

7.53 `.type` *int*

This directive, permitted only within `.def`/`.endef` pairs, records the integer *int* as the type attribute of a symbol table entry.

'`.type`' is associated only with COFF format output; when `as` is configured for `b.out` output, it accepts this directive but ignores it.

7.54 `.val` *addr*

This directive, permitted only within `.def`/`.endef` pairs, records the address *addr* as the value attribute of a symbol table entry.

'`.val`' is used only for COFF output; when `as` is configured for `b.out`, it accepts this directive but ignores it.

7.55 `.word` *expressions*

This directive expects zero or more *expressions*, of any section, separated by commas.

The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

Warning: Special Treatment to support Compilers

Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it; see Chapter 8 "Machine Dependencies," page 49), you can ignore this issue.

In order to assemble compiler output into something that works, `as` occasionlly does strange things to '`.word`' directives. Directives of the form '`.word sym1-sym2`' are often emitted by compilers as part of jump tables. Therefore, when `as` assembles a directive of the form '`.word sym1-sym2`', and the difference between `sym1` and `sym2` does not fit in 16 bits, `as` creates a *secondary jump table*, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to `sym2`. The original '`.word`' contains `sym1` minus the address of the long-jump to `sym2`.

If there were several occurrences of '`.word sym1-sym2`' before the secondary jump table, all of them are adjusted. If there was a '`.word`

`sym3-sym4`', that also did not fit in sixteen bits, a long-jump to `sym4` is included in the secondary jump table, and the `.word` directives are adjusted to contain `sym3` minus the address of the long-jump to `sym4`; and so on, for as many entries in the original jump table as necessary.

7.56 Deprecated Directives

One day these directives won't work. They are included for compatibility with older assemblers.

`.abort`

`.app-file`

`.line`

8 Machine Dependent Features

The machine instruction sets are (almost by definition) different on each machine where `as` runs. Floating point representations vary as well, and `as` often supports a few additional directives or command-line options for compatibility with other assemblers on a particular platform. Finally, some versions of `as` support special pseudo-instructions for branch optimization.

This chapter discusses most of these differences, though it does not include details on any machine's instruction set. For details on that subject, see the hardware manufacturer's manual.

8.1 VAX Dependent Features

8.1.1 VAX Command-Line Options

The Vax version of `as` accepts any of the following options, gives a warning message that the option was ignored and proceeds. These options are for compatibility with scripts designed for other people's assemblers.

`-D (Debug)`
`-S (Symbol Table)`
`-T (Token Trace)`
: These are obsolete options used to debug old assemblers.

`-d (Displacement size for JUMPs)`
: This option expects a number following the '`-d`'. Like options that expect filenames, the number may immediately follow the '`-d`' (old standard) or constitute the whole of the command line argument that follows '`-d`' (GNU standard).

`-V (Virtualize Interpass Temporary File)`
: Some other assemblers use a temporary file. This option commanded them to keep the information in active memory rather than in a disk file. `as` always does this, so this option is redundant.

`-J (JUMPify Longer Branches)`
: Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Some other

assemblers would emit short and medium branches, unless told by this option to emit short and long branches.

`-t (Temporary File Directory)`
: Some other assemblers may use a temporary file, and this option takes a filename being the directory to site the temporary file. Since `as` does not use a temporary disk file, this option makes no difference. '`-t`' needs exactly one filename.

The Vax version of the assembler accepts two options when compiled for VMS. They are '`-h`', and '`-+`'. The '`-h`' option prevents `as` from modifying the symbol-table entries for symbols that contain lowercase characters (I think). The '`-+`' option causes `as` to print warning messages if the FILENAME part of the object file, or any symbol name is larger than 31 characters. The '`-+`' option also inserts some code following the '`_main`' symbol so that the object file is compatible with Vax-11 "C".

8.1.2 VAX Floating Point

Conversion of flonums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit.

`D`, `F`, `G` and `H` floating point formats are understood.

Immediate floating literals (*e.g.* '`S'$6.9`') are rendered correctly. Again, rounding is towards zero in the boundary case.

The `.float` directive produces `f` format numbers. The `.double` directive produces `d` format numbers.

8.1.3 Vax Machine Directives

The Vax version of the assembler supports four directives for generating Vax floating point constants. They are described in the table below.

`.dfloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `d` format 64-bit floating point constants.

`.ffloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `f` format 32-bit floating point constants.

`.gfloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `g` format 64-bit floating point constants.

`.hfloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `h` format 128-bit floating point constants.

8.1.4 VAX Opcodes

All DEC mnemonics are supported. Beware that `case...` instructions have exactly 3 operands. The dispatch table that follows the `case...` instruction should be made with `.word` statements. This is compatible with all unix assemblers we know of.

8.1.5 VAX Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that reaches the target. Generally these mnemonics are made by substituting '`j`' for '`b`' at the start of a DEC mnemonic. This feature is included both for compatibility and to help compilers. If you do not need this feature, avoid these opcodes. Here are the mnemonics, and the code they can expand into.

`jbsb`
: '`Jsb`' is already an instruction mnemonic, so we chose '`jbsb`'.

  (byte displacement)
  `bsbb ...`

  (word displacement)
  `bsbw ...`

  (long displacement)
  `jsb ...`

`jbr`
`jr`
: Unconditional branch.

  (byte displacement)
  `brb ...`

  (word displacement)
  `brw ...`

  (long displacement)
  `jmp ...`

`j`*COND*
: *COND* may be any one of the conditional branches `neq`, `nequ`, `eql`, `eqlu`, `gtr`, `geq`, `lss`, `gtru`, `lequ`, `vc`, `vs`, `gequ`, `cc`, `lssu`, `cs`. *COND* may also be one of the bit tests `bs`, `bc`, `bss`, `bcs`, `bsc`, `bcc`, `bssi`, `bcci`, `lbs`, `lbc`. *NOTCOND* is the opposite condition to *COND*.

  (byte displacement)
  `b`*COND* `...`

  (word displacement)
  `b`*NOTCOND* `foo ; brw ... ; foo:`

(long displacement)

```
bNOTCOND foo ; jmp ... ; foo:
```

`jacbX` — *X* **may be one of** `b d f g h l w`.

(word displacement)

```
OPCODE ...
```

(long displacement)

```
OPCODE ..., foo ;
brb bar ;
foo: jmp ... ;
bar:
```

`jaobYYY` — *YYY* **may be one of** `lss leq`.

`jsobZZZ` — *ZZZ* **may be one of** `geq gtr`.

(byte displacement)

```
OPCODE ...
```

(word displacement)

```
OPCODE ..., foo ;
brb bar ;
foo: brw destination ;
bar:
```

(long displacement)

```
OPCODE ..., foo ;
brb bar ;
foo: jmp destination ;
bar:
```

`aobleq`
`aoblss`
`sobgeq`
`sobgtr`

(byte displacement)

```
OPCODE ...
```

(word displacement)

```
OPCODE ..., foo ;
brb bar ;
foo: brw destination ;
bar:
```

(long displacement)

```
OPCODE ..., foo ;
brb bar ;
foo: jmp destination ;
bar:
```

8.1.6 VAX Operands

The immediate character is '$' for Unix compatibility, not '#' as DEC writes it.

The indirect character is '*' for Unix compatibility, not '@' as DEC writes it.

The displacement sizing character is '`' (an accent grave) for Unix compatibility, not '^' as DEC writes it. The letter preceding '`' may have either case. 'G' is not understood, but all other letters (b i l s w) are understood.

Register names understood are r0 r1 r2 . . . r15 ap fp sp pc. Upper and lower case letters are equivalent.

For instance

```
tstb *w`$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

8.1.7 Not Supported on VAX

Vax bit fields can not be assembled with as. Someone can add the required code if they really need it.

8.2 AMD 29K Dependent Features

8.2.1 Options

`as` has no additional command-line options for the AMD 29K family.

8.2.2 Syntax

8.2.2.1 Special Characters

'`;`' is the line comment character.

'`@`' can be used instead of a newline to separate statements.

The character '`?`' is permitted in identifiers (but may not begin an identifier).

8.2.2.2 Register Names

General-purpose registers are represented by predefined symbols of the form '`GR`*nnn*' (for global registers) or '`LR`*nnn*' (for local registers), where *nnn* represents a number between `0` and `127`, written with no leading zeros. The leading letters may be in either upper or lower case; for example, '`gr13`' and '`LR7`' are both valid register names.

You may also refer to general-purpose registers by specifying the register number as the result of an expression (prefixed with '`%%`' to flag the expression as a register number):

```
%%expression
```

—where *expression* must be an absolute expression evaluating to a number between `0` and `255`. The range [0, 127] refers to global registers, and the range [128, 255] to local registers.

In addition, `as` understands the following protected special-purpose register names for the AMD 29K family:

```
vab    chd    pc0
ops    chc    pc1
cps    rbp    pc2
cfg    tmc    mmu
cha    tmr    lru
```

These unprotected special-purpose register names are also recognized:

```
ipc    alu    fpe
ipa    bp     inte
ipb    fc     fps
q      cr     exop
```

8.2.3 Floating Point

The AMD 29K family uses IEEE floating-point numbers.

8.2.4 AMD 29K Machine Directives

`.block` *size* , *fill*
: This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

  In other versions of the GNU assembler, this directive is called '`.space`'.

`.cputype`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.file`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

  > *Warning:* in other versions of the GNU assembler, `.file` is used for the directive called `.app-file` in the AMD 29K support.

`.line`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.sect`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.use` *section name*
: Establishes the section and subsection for the following code; *section name* may be one of `.text`, `.data`, `.data1`, or `.lit`. With one of the first three *section name* options, '`.use`' is equivalent to the machine directive *section name*; the remaining case, '`.use .lit`', is the same as '`.data 200`'.

8.2.5 Opcodes

`as` implements all the standard AMD 29K opcodes. No additional pseudo-instructions are needed on this family.

For information on the 29K machine instruction set, see *Am29000 User's Manual*, Advanced Micro Devices, Inc.

8.3 H8/300 Dependent Features

8.3.1 Options

`as` has no additional command-line options for the Hitachi H8/300 family.

8.3.2 Syntax

8.3.2.1 Special Characters

'`;`' is the line comment character.

'`$`' can be used instead of a newline to separate statements. Therefore *you may not use '`$`' in symbol names* on the H8/300.

8.3.2.2 Register Names

You can use predefined symbols of the form '`rnh`' and '`rnl`' to refer to the H8/300 registers as sixteen 8-bit general-purpose registers. *n* is a digit from '`0`' to '`7`'); for instance, both '`r0h`' and '`r7l`' are valid register names.

You can also use the eight predefined symbols '`rn`' to refer to the H8/300 registers as 16-bit registers (you must use this form for addressing).

On the H8/300H, you can also use the eight predefined symbols '`ern`' ('`er0`' . . . '`er7`') to refer to the 32-bit general purpose registers.

The two control registers are called `pc` (program counter; a 16-bit register, except on the H8/300H where it is 24 bits) and `ccr` (condition code register; an 8-bit register). `r7` is used as the stack pointer, and can also be called `sp`.

8.3.2.3 Addressing Modes

as understands the following addressing modes for the H8/300:

`rn` Register direct

`@rn` Register indirect

`@(d, rn)`
`@(d:16, rn)`
`@(d:24, rn)`
: Register indirect: 16-bit or 24-bit displacement *d* from register *n*. (24-bit displacements are only meaningful on the H8/300H.)

`@rn+` Register indirect with post-increment

`@-rn` Register indirect with pre-decrement

`@aa`
`@aa:8`
`@aa:16`
`@aa:24` Absolute address `aa`. (The address size '`:24`' only makes sense on the H8/300H.)

`#xx`
`#xx:8`
`#xx:16`
`#xx:32` Immediate data *xx*. You may specify the '`:8`', '`:16`', or '`:32`' for clarity, if you wish; but `as` neither requires this nor uses it—the data size required is taken from context.

`@@aa`
`@@aa:8` Memory indirect. You may specify the '`:8`' for clarity, if you wish; but `as` neither requires this nor uses it.

8.3.3 Floating Point

The H8/300 family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

8.3.4 H8/300 Machine Directives

`as` has only one machine-dependent directive for the H8/300:

`.h8300h` Recognize and emit additional instructions for the H8/300H variant, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

On the H8/300 family (including the H8/300H) '`.word`' directives generate 16-bit numbers.

8.3.5 Opcodes

For detailed information on the H8/300 machine instruction set, see *H8/300 Series Programming Manual* (Hitachi ADE–602–025). For information specific to the H8/300H, see *H8/300H Series Programming Manual* (Hitachi).

`as` implements all the standard H8/300 opcodes. No additional pseudo-instructions are needed on this family.

The following table summarizes the H8/300 opcodes, and their arguments. Entries marked '`*`' are opcodes used only on the H8/300H.

Legend:

`Rs`	source register
`Rd`	destination register
`abs`	absolute address
`imm`	immediate data
`disp:N`	N-bit displacement from a register
`pcrel:N`	N-bit displacement relative to program counter

```
     add.b #imm,rd              *  andc #imm,ccr
     add.b rs,rd                   band #imm,rd
     add.w rs,rd                   band #imm,@rd
  *  add.w #imm,rd                 band #imm,@abs:8
  *  add.l rs,rd                   bra  pcrel:8
  *  add.l #imm,rd              *  bra  pcrel:16
     adds #imm,rd                  bt   pcrel:8
     addx #imm,rd               *  bt   pcrel:16
     addx rs,rd                    brn  pcrel:8
     and.b #imm,rd              *  brn  pcrel:16
     and.b rs,rd                   bf   pcrel:8
  *  and.w rs,rd                *  bf   pcrel:16
  *  and.w #imm,rd                 bhi  pcrel:8
  *  and.l #imm,rd              *  bhi  pcrel:16
  *  and.l rs,rd                   bls  pcrel:8
```

```
*  bls  pcrel:16
   bcc  pcrel:8
*  bcc  pcrel:16
   bhs  pcrel:8
*  bhs  pcrel:16
   bcs  pcrel:8
*  bcs  pcrel:16
   blo  pcrel:8
*  blo  pcrel:16
   bne  pcrel:8
*  bne  pcrel:16
   beq  pcrel:8
*  beq  pcrel:16
   bvc  pcrel:8
*  bvc  pcrel:16
   bvs  pcrel:8
*  bvs  pcrel:16
   bpl  pcrel:8
*  bpl  pcrel:16
   bmi  pcrel:8
*  bmi  pcrel:16
   bge  pcrel:8
*  bge  pcrel:16
   blt  pcrel:8
*  blt  pcrel:16
   bgt  pcrel:8
*  bgt  pcrel:16
   ble  pcrel:8
*  ble  pcrel:16
   bclr #imm,rd
   bclr #imm,@rd
   bclr #imm,@abs:8
   bclr rs,rd
   bclr rs,@rd
   bclr rs,@abs:8
   biand #imm,rd
   biand #imm,@rd
   biand #imm,@abs:8
   bild #imm,rd
   bild #imm,@rd
   bild #imm,@abs:8
   bior #imm,rd
   bior #imm,@rd
   bior #imm,@abs:8
   bist #imm,rd
   bist #imm,@rd
   bist #imm,@abs:8
   bixor #imm,rd
   bixor #imm,@rd
   bixor #imm,@abs:8
   bld  #imm,rd
   bld  #imm,@rd
   bld  #imm,@abs:8
   bnot #imm,rd
   bnot #imm,@rd
   bnot #imm,@abs:8
   bnot rs,rd
   bnot rs,@rd
   bnot rs,@abs:8
   bor  #imm,rd
   bor  #imm,@rd
   bor  #imm,@abs:8
   bset #imm,rd
   bset #imm,@rd
   bset #imm,@abs:8
   bset rs,rd
   bset rs,@rd
   bset rs,@abs:8
   bsr  pcrel:8
   bsr  pcrel:16
   bst  #imm,rd
   bst  #imm,@rd
   bst  #imm,@abs:8
   btst #imm,rd
   btst #imm,@rd
   btst #imm,@abs:8
   btst rs,rd
   btst rs,@rd
   btst rs,@abs:8
   bxor #imm,rd
   bxor #imm,@rd
   bxor #imm,@abs:8
   cmp.b #imm,rd
   cmp.b rs,rd
   cmp.w rs,rd
   cmp.w rs,rd
*  cmp.w #imm,rd
*  cmp.l #imm,rd
*  cmp.l rs,rd
   daa  rs
   das  rs
   dec.b rs
*  dec.w #imm,rd
*  dec.l #imm,rd
   divxu.b rs,rd
*  divxu.w rs,rd
*  divxs.b rs,rd
*  divxs.w rs,rd
   eepmov
*  eepmovw
```

```
*  exts.w rd
*  exts.l rd
*  extu.w rd
*  extu.l rd
   inc  rs
*  inc.w #imm,rd
*  inc.l #imm,rd
   jmp  @rs
   jmp  abs
   jmp  @@abs:8
   jsr  @rs
   jsr  abs
   jsr  @@abs:8
   ldc  #imm,ccr
   ldc  rs,ccr
*  ldc  @abs:16,ccr
*  ldc  @abs:24,ccr
*  ldc  @(disp:16,rs),ccr
*  ldc  @(disp:24,rs),ccr
*  ldc  @rs+,ccr
*  ldc  @rs,ccr
*  mov.b @(disp:24,rs),rd
*  mov.b rs,@(disp:24,rd)
   mov.b @abs:16,rd
   mov.b rs,rd
   mov.b @abs:8,rd
   mov.b rs,@abs:8
   mov.b rs,rd
   mov.b #imm,rd
   mov.b @rs,rd
   mov.b @(disp:16,rs),rd
   mov.b @rs+,rd
   mov.b @abs:8,rd
   mov.b rs,@rd
   mov.b rs,@(disp:16,rd)
   mov.b rs,@-rd
   mov.b rs,@abs:8
   mov.w rs,@rd
*  mov.w @(disp:24,rs),rd
*  mov.w rs,@(disp:24,rd)
*  mov.w @abs:24,rd
*  mov.w rs,@abs:24
   mov.w rs,rd
   mov.w #imm,rd
   mov.w @rs,rd
   mov.w @(disp:16,rs),rd
   mov.w @rs+,rd
   mov.w @abs:16,rd
   mov.w rs,@(disp:16,rd)
   mov.w rs,@-rd
   mov.w rs,@abs:16
*  mov.l #imm,rd
*  mov.l rs,rd
*  mov.l @rs,rd
*  mov.l @(disp:16,rs),rd
*  mov.l @(disp:24,rs),rd
*  mov.l @rs+,rd
*  mov.l @abs:16,rd
*  mov.l @abs:24,rd
*  mov.l rs,@rd
*  mov.l rs,@(disp:16,rd)
*  mov.l rs,@(disp:24,rd)
*  mov.l rs,@-rd
*  mov.l rs,@abs:16
*  mov.l rs,@abs:24
   movfpe @abs:16,rd
   movtpe rs,@abs:16
   mulxu.b rs,rd
*  mulxu.w rs,rd
*  mulxs.b rs,rd
*  mulxs.w rs,rd
   neg.b rs
*  neg.w rs
*  neg.l rs
   nop
   not.b rs
*  not.w rs
*  not.l rs
   or.b #imm,rd
   or.b rs,rd
*  or.w #imm,rd
*  or.w rs,rd
*  or.l #imm,rd
*  or.l rs,rd
   orc  #imm,ccr
   pop.w rs
*  pop.l rs
   push.w rs
*  push.l rs
   rotl.b rs
*  rotl.w rs
*  rotl.l rs
   rotr.b rs
*  rotr.w rs
*  rotr.l rs
   rotxl.b rs
*  rotxl.w rs
*  rotxl.l rs
   rotxr.b rs
*  rotxr.w rs
```

```
*  rotxr.l rs               *  stc  ccr,@(disp:24,rd)
   bpt                      *  stc  ccr,@-rd
   rte                      *  stc  ccr,@abs:16
   rts                      *  stc  ccr,@abs:24
   shal.b rs                   sub.b rs,rd
*  shal.w rs                   sub.w rs,rd
*  shal.l rs                *  sub.w #imm,rd
   shar.b rs                *  sub.l rs,rd
*  shar.w rs                *  sub.l #imm,rd
*  shar.l rs                   subs #imm,rd
   shll.b rs                   subx #imm,rd
*  shll.w rs                   subx rs,rd
*  shll.l rs                *  trapa #imm
   shlr.b rs                   xor  #imm,rd
*  shlr.w rs                   xor  rs,rd
*  shlr.l rs                *  xor.w #imm,rd
   sleep                    *  xor.w rs,rd
   stc  ccr,rd              *  xor.l #imm,rd
*  stc  ccr,@rs             *  xor.l rs,rd
*  stc  ccr,@(disp:16,rd)      xorc #imm,ccr
```

Four H8/300 instructions (`add`, `cmp`, `mov`, `sub`) are defined with variants using the suffixes '`.b`', '`.w`', and '`.l`' to specify the size of a memory operand. `as` supports these suffixes, but does not require them; since one of the operands is always a register, `as` can deduce the correct size.

For example, since `r0` refers to a 16-bit register,

```
mov    r0,@foo
```

is equivalent to

```
mov.w  r0,@foo
```

If you use the size suffixes, `as` issues a warning when the suffix and the register size do not match.

8.4 H8/500 Dependent Features

8.4.1 Options

`as` has no additional command-line options for the Hitachi H8/500 family.

8.4.2 Syntax

8.4.2.1 Special Characters

'`!`' is the line comment character.

'`;`' can be used instead of a newline to separate statements.

Since '`$`' has no special meaning, you may use it in symbol names.

8.4.2.2 Register Names

You can use the predefined symbols '`r0`', '`r1`', '`r2`', '`r3`', '`r4`', '`r5`', '`r6`', and '`r7`' to refer to the H8/500 registers.

The H8/500 also has these control registers:

`cp` code pointer

`dp` data pointer

`bp` base pointer

`tp` stack top pointer

`ep` extra pointer

`sr` status register

`ccr` condition code register

All registers are 16 bits long. To represent 32 bit numbers, use two adjacent registers; for distant memory addresses, use one of the segment pointers (`cp` for the program counter; `dp` for `r0`–`r3`; `ep` for `r4` and `r5`; and `tp` for `r6` and `r7`.

8.4.2.3 Addressing Modes

as understands the following addressing modes for the H8/500:

`R`*n* Register direct

`@R`*n* Register indirect

`@(d:8, R`*n*`)`
Register indirect with 8 bit signed displacement

`@(d:16, R`*n*`)`
Register indirect with 16 bit signed displacement

`@-R`*n* Register indirect with pre-decrement

`@R`*n*`+` Register indirect with post-increment

`@`*aa*`:8` 8 bit absolute address

`@`*aa*`:16` 16 bit absolute address

`#`*xx*`:8` 8 bit immediate

`#`*xx*`:16` 16 bit immediate

8.4.3 Floating Point

The H8/500 family uses IEEE floating-point numbers.

8.4.4 H8/500 Machine Directives

`as` has no machine-dependent directives for the H8/500. However, on this platform the '`.int`' and '`.word`' directives generate 16-bit numbers.

8.4.5 Opcodes

For detailed information on the H8/500 machine instruction set, see *H8/500 Series Programming Manual* (Hitachi M21T001).

`as` implements all the standard H8/500 opcodes. No additional pseudo-instructions are needed on this family.

The following table summarizes H8/500 opcodes and their operands:

Legend:

`abs8`	8-bit absolute address
`abs16`	16-bit absolute address
`abs24`	24-bit absolute address
`crb`	`ccr`, `br`, `ep`, `dp`, `tp`, `dp`
`disp8`	8-bit displacement
`ea`	`rn`, `@rn`, `@(d:8, rn)`, `@(d:16, rn)`, `@-rn`, `@rn+`, `@aa:8`, `@aa:16`, `#xx:8`, `#xx:16`
`ea_mem`	`@rn`, `@(d:8, rn)`, `@(d:16, rn)`, `@-rn`, `@rn+`, `@aa:8`, `@aa:16`
`ea_noimm`	`rn`, `@rn`, `@(d:8, rn)`, `@(d:16, rn)`, `@-rn`, `@rn+`, `@aa:8`, `@aa:16`
`fp`	`r6`
`imm4`	4-bit immediate data
`imm8`	8-bit immediate data
`imm16`	16-bit immediate data
`pcrel8`	8-bit offset from program counter
`pcrel16`	16-bit offset from program counter
`qim`	`-2`, `-1`, `1`, `2`
`rd`	any register
`rs`	a register distinct from rd
`rlist`	comma-separated list of registers in parentheses; register ranges `rd-rs` are allowed
`sp`	stack pointer (`r7`)
`sr`	status register
`sz`	size; '`.b`' or '`.w`'. If omitted, default '`.w`'

```
ldc[.b] ea,crb
ldc[.w] ea,sr
add[:q] sz qim,ea_noimm
add[:g] sz ea,rd
adds sz ea,rd
addx sz ea,rd
and sz ea,rd
andc[.b] imm8,crb
andc[.w] imm16,sr
bpt
bra[.w] pcrel16
bra[.b] pcrel8
bt[.w] pcrel16
bt[.b] pcrel8
brn[.w] pcrel16
brn[.b] pcrel8
bf[.w] pcrel16
bf[.b] pcrel8
bhi[.w] pcrel16
bhi[.b] pcrel8
bls[.w] pcrel16
bls[.b] pcrel8
bcc[.w] pcrel16
bcc[.b] pcrel8
bhs[.w] pcrel16
bhs[.b] pcrel8
bcs[.w] pcrel16
bcs[.b] pcrel8
blo[.w] pcrel16
blo[.b] pcrel8
bne[.w] pcrel16
bne[.b] pcrel8
beq[.w] pcrel16
beq[.b] pcrel8
bvc[.w] pcrel16
bvc[.b] pcrel8
bvs[.w] pcrel16
bvs[.b] pcrel8
bpl[.w] pcrel16
bpl[.b] pcrel8
bmi[.w] pcrel16
bmi[.b] pcrel8
bge[.w] pcrel16
bge[.b] pcrel8
```

```
blt[.w] pcrel16
blt[.b] pcrel8
bgt[.w] pcrel16
bgt[.b] pcrel8
ble[.w] pcrel16
ble[.b] pcrel8
bclr sz imm4,ea_noimm
bclr sz rs,ea_noimm
bnot sz imm4,ea_noimm
bnot sz rs,ea_noimm
bset sz imm4,ea_noimm
bset sz rs,ea_noimm
bsr[.b] pcrel8
bsr[.w] pcrel16
btst sz imm4,ea_noimm
btst sz rs,ea_noimm
clr sz ea
cmp[:e][.b] imm8,rd
cmp[:i][.w] imm16,rd
cmp[:g].b imm8,ea_noimm
cmp[:g][.w] imm16,ea_noimm
Cmp[:g] sz ea,rd
dadd rs,rd
divxu sz ea,rd
dsub rs,rd
exts[.b] rd
extu[.b] rd
jmp @rd
jmp @(imm8,rd)
jmp @(imm16,rd)
jmp abs16
jsr @rd
jsr @(imm8,rd)
jsr @(imm16,rd)
jsr abs16
ldm @sp+,(rlist)
link fp,imm8
link fp,imm16
mov[:e][.b] imm8,rd
mov[:i][.w] imm16,rd
mov[:l][.w] abs8,rd
mov[:l].b abs8,rd
mov[:s][.w] rs,abs8
mov[:s].b rs,abs8
mov[:f][.w] @(disp8,fp),rd
mov[:f][.w] rs,@(disp8,fp)
mov[:f].b @(disp8,fp),rd
mov[:f].b rs,@(disp8,fp)
mov[:g] sz rs,ea_mem
mov[:g] sz ea,rd
mov[:g][.b] imm8,ea_mem
mov[:g][.w] imm16,ea_mem
movfpe[.b] ea,rd
movtpe[.b] rs,ea_noimm
mulxu sz ea,rd
neg sz ea
nop
not sz ea
or sz ea,rd
orc[.b] imm8,crb
orc[.w] imm16,sr
pjmp abs24
pjmp @rd
pjsr abs24
pjsr @rd
prtd imm8
prtd imm16
prts
rotl sz ea
rotr sz ea
rotxl sz ea
rotxr sz ea
rtd imm8
rtd imm16
rts
scb/f rs,pcrel8
scb/ne rs,pcrel8
scb/eq rs,pcrel8
shal sz ea
shar sz ea
shll sz ea
shlr sz ea
sleep
stc[.b] crb,ea_noimm
stc[.w] sr,ea_noimm
stm (rlist),@-sp
sub sz ea,rd
subs sz ea,rd
subx sz ea,rd
swap[.b] rd
tas[.b] ea
trapa imm4
trap/vs
tst sz ea
unlk fp
xch[.w] rs,rd
xor sz ea,rd
xorc.b imm8,crb
xorc.w imm16,sr
```

8.5 HPPA Dependent Features

8.5.1 Notes

As a back end for GNU CC as has been throughly tested and should work extremely well. We have tested it only minimally on hand written assembly code and no one has tested it much on the assembly output from the HP compilers.

The format of the debugging sections has changed since the original as port (version 1.3X) was released; therefore, you must rebuild all HPPA objects and libraries with the new assembler so that you can debug the final executable.

The HPPA as port generates a small subset of the relocations available in the SOM and ELF object file formats. Additional relocation support will be added as it becomes necessary.

8.5.2 Options

as has no machine-dependent command-line options for the HPPA.

8.5.3 Syntax

The assembler syntax closely follows the HPPA instruction set reference manual; assembler directives and general syntax closely follow the HPPA assembly language reference manual, with a few noteworthy differences.

First, a colon may immediately follow a label definition. This is simply for compatibility with how most assembly language programmers write code.

Some obscure expression parsing problems may affect hand written code which uses the spop instructions, or code which makes significant use of the ! line separator.

as is much less forgiving about missing arguments and other similar oversights than the HP assembler. as notifies you of missing arguments as syntax errors; this is regarded as a feature, not a bug.

Finally, as allows you to use an external symbol without explicitly importing the symbol. *Warning:* in the future this will be an error for HPPA targets.

Special characters for HPPA targets include:

'`;`' is the line comment character.

'`!`' can be used instead of a newline to separate statements.

Since '`$`' has no special meaning, you may use it in symbol names.

8.5.4 Floating Point

The HPPA family uses IEEE floating-point numbers.

8.5.5 HPPA Assembler Directives

`as` for the HPPA supports many additional directives for compatibility with the native assembler. This section describes them only briefly. For detailed information on HPPA-specific assembler directives, see *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001).

`as` does *not* support the following assembler directives described in the HP manual:

```
.endm           .liston
.enter          .locct
.leave          .macro
.listoff
```

Beyond those implemented for compatibility, `as` supports one additional assembler directive for the HPPA: `.param`. It conveys register argument locations for static functions. Its syntax closely follows the `.export` directive.

These are the additional directives in `as` for the HPPA:

`.block` *n*
`.blockz` *n*
: Reserve *n* bytes of storage, and initialize them to zero.

`.call`
: Mark the beginning of a procedure call. Only the special case with *no arguments* is allowed.

`.callinfo [ param=value, ... ] [ flag, ... ]`
: Specify a number of parameters and flags that define the environment for a procedure.

  param may be any of '`frame`' (frame size), '`entry_gr`' (end of general register range), '`entry_fr`' (end of float register range), '`entry_sr`' (end of space register range).

  The values for *flag* are '`calls`' or '`caller`' (proc has subroutines), '`no_calls`' (proc does not call subroutines), '`save_rp`' (preserve return pointer), '`save_sp`' (proc preserves stack pointer), '`no_unwind`' (do not unwind this proc), '`hpux_int`' (proc is interrupt routine).

`.code`
: Assemble into the standard section called '`$TEXT$`', subsection '`$CODE$`'.

`.copyright "string"`
: In the SOM object format, insert *string* into the object code, marked as a copyright string.

`.copyright "string"`
: In the ELF object format, insert *string* into the object code, marked as a version string.

`.enter` Not yet supported; the assembler rejects programs containing this directive.

`.entry` Mark the beginning of a procedure.

`.exit` Mark the end of a procedure.

`.export name [ ,typ ] [ ,param=r ]`
: Make a procedure *name* available to callers. *typ*, if present, must be one of 'absolute', 'code' (ELF only, not SOM), 'data', 'entry', 'data', 'entry', 'millicode', 'plabel', 'pri_prog', or 'sec_prog'.

param, if present, provides either relocation information for the procedure arguments and result, or a privilege level. *param* may be 'argw*n*' (where *n* ranges from 0 to 3, and indicates one of four one-word arguments); 'rtnval' (the procedure's result); or 'priv_lev' (privilege level). For arguments or the result, *r* specifies how to relocate, and must be one of 'no' (not relocatable), 'gr' (argument is in general register), 'fr' (in floating point register), or 'fu' (upper half of float register). For 'priv_lev', *r* is an integer.

`.half n` Define a two-byte integer constant *n*; synonym for the portable as directive .short.

`.import name [ ,typ ]`
: Converse of .export; make a procedure available to call. The arguments use the same conventions as the first two arguments for .export.

`.label name`
: Define *name* as a label for the current assembly location.

`.leave` Not yet supported; the assembler rejects programs containing this directive.

`.origin lc`
: Advance location counter to *lc*. Synonym for the {No value for ``as''} portable directive .org.

`.param name [ ,typ ] [ ,param=r ]`
: Similar to .export, but used for static procedures.

`.proc` Use preceding the first statement of a procedure.

`.procend` Use following the last statement of a procedure.

label `.reg` *expr*
: Synonym for `.equ`; define *label* with the absolute expression *expr* as its value.

`.space` *secname* `[ ,`*params* `]`
: Switch to section *secname*, creating a new section by that name if necessary. You may only use *params* when creating a new section, not when switching to an existing one. *secname* may identify a section by number rather than by name.

  If specified, the list *params* declares attributes of the section, identified by keywords. The keywords recognized are '`spnum=`*exp*' (identify this section by the number *exp*, an absolute expression), '`sort=`*exp*' (order sections according to this sort key when linking; *exp* is an absolute expression), '`unloadable`' (section contains no loadable data), '`notdefined`' (this section defined elsewhere), and '`private`' (data in this section not available to other programs).

`.spnum` *secnam*
: Allocate four bytes of storage, and initialize them with the section number of the section named *secnam*. (You can define the section number with the HPPA `.space` directive.)

`.string "`*str*`"`
: Copy the characters in the string *str* to the object file. See Section 3.6.1.1 "Strings," page 16, for information on escape sequences you can use in `as` strings.

  Warning! The HPPA version of `.string` differs from the usual `as` definition: it does *not* write a zero byte after copying *str*.

`.stringz "`*str*`"`
: Like `.string`, but appends a zero byte after copying *str* to object file.

`.subspa` *name* `[ ,`*params* `]`
: Similar to `.space`, but selects a subsection *name* within the current section. You may only specify *params* when you create a subsection (in the first instance of `.subspa` for this *name*).

  If specified, the list *params* declares attributes of the subsection, identified by keywords. The keywords recognized are '`quad=`*expr*' ("quadrant" for this subsection), '`align=`*expr*' (alignment for beginning of this subsection; a power of two),

'`access=expr`' (value for "access rights" field), '`sort=expr`' (sorting order for this subspace in link), '`code_only`' (subsection contains only code), '`unloadable`' (subsection cannot be loaded into memory), '`common`' (subsection is common block), '`dup_comm`' (initialized data may have duplicate names), or '`zero`' (subsection is all zeros, do not write in object file).

`.version "str"`
Write *str* as version identifier in object code.

8.5.6 Opcodes

For detailed information on the HPPA machine instruction set, see *PA-RISC Architecture and Instruction Set Reference Manual* (HP 09740-90039).

8.6 Hitachi SH Dependent Features

8.6.1 Options

`as` has no additional command-line options for the Hitachi SH family.

8.6.2 Syntax

8.6.2.1 Special Characters

'`!`' is the line comment character.

You can use '`;`' instead of a newline to separate statements.

Since '`$`' has no special meaning, you may use it in symbol names.

8.6.2.2 Register Names

You can use the predefined symbols '`r0`', '`r1`', '`r2`', '`r3`', '`r4`', '`r5`', '`r6`', '`r7`', '`r8`', '`r9`', '`r10`', '`r11`', '`r12`', '`r13`', '`r14`', and '`r15`' to refer to the SH registers.

The SH also has these control registers:

`pr` procedure register (holds return address)

`pc` program counter

`mach`
`macl` high and low multiply accumulator registers

`sr` status register

`gbr` global base register

`vbr` vector base register (for interrupt vectors)

8.6.2.3 Addressing Modes

`as` understands the following addressing modes for the SH. `R`*n* in the following refers to any of the numbered registers, but *not* the control registers.

`R`*n* Register direct

`@R`*n* Register indirect

`@-R`*n* Register indirect with pre-decrement

`@Rn+` Register indirect with post-increment

`@(disp, Rn)`
Register indirect with displacement

`@(R0, Rn)`
Register indexed

`@(disp, GBR)`
`GBR` offset

`@(R0, GBR)`
GBR indexed

addr
`@(disp, PC)`
PC relative address (for branch or for addressing memory). The `as` implementation allows you to use the simpler form *addr* anywhere a PC relative address is called for; the alternate form is supported for compatibility with other assemblers.

`#imm` Immediate data

8.6.3 Floating Point

The SH family uses IEEE floating-point numbers.

8.6.4 SH Machine Directives

`as` has no machine-dependent directives for the SH.

8.6.5 Opcodes

For detailed information on the SH machine instruction set, see *SH-Microcomputer User's Manual* (Hitachi Micro Systems, Inc.).

`as` implements all the standard SH opcodes. No additional pseudo-instructions are needed on this family. Note, however, that because `as` supports a simpler form of PC-relative addressing, you may simply write (for example)

```
mov.l  bar,r0
```

where other assemblers might require an explicit displacement to `bar` from the program counter:

```
mov.l  @(disp, PC)
```

Here is a summary of SH opcodes:

Legend:

`Rn`	a numbered register
`Rm`	another numbered register
`#imm`	immediate data
`disp`	displacement
`disp8`	8-bit displacement
`disp12`	12-bit displacement

```
add #imm,Rn
add Rm,Rn
addc Rm,Rn
addv Rm,Rn
and #imm,R0
and Rm,Rn
and.b #imm,@(R0,GBR)
bf disp8
bra disp12
bsr disp12
bt disp8
clrmac
clrt
cmp/eq #imm,R0
cmp/eq Rm,Rn
cmp/ge Rm,Rn
cmp/gt Rm,Rn
cmp/hi Rm,Rn
cmp/hs Rm,Rn
cmp/pl Rn
cmp/pz Rn
cmp/str Rm,Rn
div0s Rm,Rn
div0u
div1 Rm,Rn
exts.b Rm,Rn
exts.w Rm,Rn
extu.b Rm,Rn
extu.w Rm,Rn
jmp @Rn
jsr @Rn
ldc Rn,GBR
ldc Rn,SR
ldc Rn,VBR
ldc.l @Rn+,GBR
ldc.l @Rn+,SR
ldc.l @Rn+,VBR
lds Rn,MACH
lds Rn,MACL
lds Rn,PR
lds.l @Rn+,MACH
lds.l @Rn+,MACL
lds.l @Rn+,PR
mac.w @Rm+,@Rn+
mov #imm,Rn
mov Rm,Rn
mov.b Rm,@(R0,Rn)
mov.b Rm,@-Rn
mov.b Rm,@Rn
mov.b @(disp,Rm),R0
mov.b @(disp,GBR),R0
mov.b @(R0,Rm),Rn
mov.b @Rm+,Rn
mov.b @Rm,Rn
mov.b R0,@(disp,Rm)
mov.b R0,@(disp,GBR)
mov.l Rm,@(disp,Rn)
mov.l Rm,@(R0,Rn)
mov.l Rm,@-Rn
mov.l Rm,@Rn
mov.l @(disp,Rn),Rm
mov.l @(disp,GBR),R0
mov.l @(disp,PC),Rn
mov.l @(R0,Rm),Rn
mov.l @Rm+,Rn
mov.l @Rm,Rn
mov.l R0,@(disp,GBR)
mov.w Rm,@(R0,Rn)
mov.w Rm,@-Rn
mov.w Rm,@Rn
mov.w @(disp,Rm),R0
mov.w @(disp,GBR),R0
mov.w @(disp,PC),Rn
mov.w @(R0,Rm),Rn
mov.w @Rm+,Rn
mov.w @Rm,Rn
mov.w R0,@(disp,Rm)
mov.w R0,@(disp,GBR)
mova @(disp,PC),R0
movt Rn
muls Rm,Rn
mulu Rm,Rn
neg Rm,Rn
negc Rm,Rn
```

```
nop
not Rm,Rn
or #imm,R0
or Rm,Rn
or.b #imm,@(R0,GBR)
rotcl Rn
rotcr Rn
rotl Rn
rotr Rn
rte
rts
sett
shal Rn
shar Rn
shll Rn
shll16 Rn
shll2 Rn
shll8 Rn
shlr Rn
shlr16 Rn
shlr2 Rn
shlr8 Rn
sleep
stc GBR,Rn
stc SR,Rn
stc VBR,Rn
stc.l GBR,@-Rn
stc.l SR,@-Rn
stc.l VBR,@-Rn
sts MACH,Rn
sts MACL,Rn
sts PR,Rn
sts.l MACH,@-Rn
sts.l MACL,@-Rn
sts.l PR,@-Rn
sub Rm,Rn
subc Rm,Rn
subv Rm,Rn
swap.b Rm,Rn
swap.w Rm,Rn
tas.b @Rn
trapa #imm
tst #imm,R0
tst Rm,Rn
tst.b #imm,@(R0,GBR)
xor #imm,R0
xor Rm,Rn
xor.b #imm,@(R0,GBR)
xtrct Rm,Rn
```

8.7 Intel 80960 Dependent Features

8.7.1 i960 Command-line Options

-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC
: Select the 80960 architecture. Instructions or features not supported by the selected architecture cause fatal errors.

'-ACA' is equivalent to '-ACA_A'; '-AKC' is equivalent to '-AMC'. Synonyms are provided for compatibility with other tools.

If you do not specify any of these options, as generates code for any instruction or feature that is supported by *some* version of the 960 (even if this means mixing architectures!). In principle, as attempts to deduce the minimal sufficient processor type if none is specified; depending on the object code format, the processor type may be recorded in the object file. If it is critical that the as output match a specific architecture, specify that architecture explicitly.

-b
: Add code to collect information about conditional branches taken, for later optimization using branch prediction bits. (The conditional branch instructions have branch prediction bits in the CA, CB, and CC architectures.) If *BR* represents a conditional branch instruction, the following represents the code generated by the assembler when '-b' is specified:

```
        call    increment routine
        .word   0       # pre-counter
Label:  BR
        call    increment routine
        .word   0       # post-counter
```

The counter following a branch records the number of times that branch was *not* taken; the differenc between the two counters is the number of times the branch *was* taken.

A table of every such Label is also generated, so that the external postprocessor gbr960 (supplied by Intel) can locate all the counters. This table is always labelled '__BRANCH_TABLE__'; this is a local symbol to permit collecting statistics for many separate object files. The table is word aligned, and begins with a two-word header. The first word, initialized to 0, is used in maintaining linked lists of branch tables. The second word is a count of the number of entries in the table, which follow immediately: each is a word, pointing to one of the labels illustrated above.

__BRANCH_TABLE__ *layout*

The first word of the header is used to locate multiple branch tables, since each object file may contain one. Normally the links are maintained with a call to an initialization routine, placed at the beginning of each function in the file. The GNU C compiler generates these calls automatically when you give it a '-b' option. For further details, see the documentation of 'gbr960'.

-no-relax
: Normally, Compare-and-Branch instructions with targets that require displacements greater than 13 bits (or that have external targets) are replaced with the corresponding compare (or 'chkbit') and branch instructions. You can use the '-no-relax' option to specify that as should generate errors instead, if the target displacement is larger than 13 bits.

  This option does not affect the Compare-and-Jump instructions; the code emitted for them is *always* adjusted when necessary (depending on displacement size), regardless of whether you use '-no-relax'.

8.7.2 Floating Point

as generates IEEE floating-point numbers for the directives '.float', '.double', '.extended', and '.single'.

8.7.3 i960 Machine Directives

.bss *symbol*, *length*, *align*
: Reserve *length* bytes in the bss section for a local *symbol*, aligned to the power of two specified by *align*. *length* and *align* must be positive absolute expressions. This directive differs from '.lcomm' only in that it permits you to specify an alignment. See Section 7.29 ".lcomm," page 40.

.extended *flonums*
: .extended expects zero or more flonums, separated by commas; for each flonum, '.extended' emits an IEEE extended-format (80-bit) floating-point number.

`.leafproc` *call-lab*, *bal-lab*

You can use the '`.leafproc`' directive in conjunction with the optimized `callj` instruction to enable faster calls of leaf procedures. If a procedure is known to call no other procedures, you may define an entry point that skips procedure prolog code (and that does not depend on system-supplied saved context), and declare it as the *bal-lab* using '`.leafproc`'. If the procedure also has an entry point that goes through the normal prolog, you can specify that entry point as *call-lab*.

A '`.leafproc`' declaration is meant for use in conjunction with the optimized call instruction '`callj`'; the directive records the data needed later to choose between converting the '`callj`' into a `bal` or a `call`.

call-lab is optional; if only one argument is present, or if the two arguments are identical, the single argument is assumed to be the `bal` entry point.

`.sysproc` *name*, *index*

The '`.sysproc`' directive defines a name for a system procedure. After you define it using '`.sysproc`', you can use *name* to refer to the system procedure identified by *index* when calling procedures with the optimized call instruction '`callj`'.

Both arguments are required; *index* must be between 0 and 31 (inclusive).

8.7.4 i960 Opcodes

All Intel 960 machine instructions are supported; see Section 8.7.1 "i960 Command-line Options," page 75 for a discussion of selecting the instruction subset for a particular 960 architecture.

Some opcodes are processed beyond simply emitting a single corresponding instruction: '`callj`', and Compare-and-Branch or Compare-and-Jump instructions with target displacements larger than 13 bits.

8.7.4.1 `callj`

You can write `callj` to have the assembler or the linker determine the most appropriate form of subroutine call: '`call`', '`bal`', or '`calls`'. If the assembly source contains enough information—a '`.leafproc`' or '`.sysproc`' directive defining the operand—then `as` translates the `callj`; if not, it simply emits the `callj`, leaving it for the linker to resolve.

8.7.4.2 Compare-and-Branch

The 960 architectures provide combined Compare-and-Branch instructions that permit you to store the branch target in the lower 13 bits of the instruction word itself. However, if you specify a branch target far enough away that its address won't fit in 13 bits, the assembler can either issue an error, or convert your Compare-and-Branch instruction into separate instructions to do the compare and the branch.

Whether `as` gives an error or expands the instruction depends on two choices you can make: whether you use the '`-no-relax`' option, and whether you use a "Compare and Branch" instruction or a "Compare and Jump" instruction. The "Jump" instructions are *always* expanded if necessary; the "Branch" instructions are expanded when necessary *unless* you specify `-no-relax`—in which case `as` gives an error instead.

These are the Compare-and-Branch instructions, their "Jump" variants, and the instruction pairs they may expand into:

Compare and		
Branch	*Jump*	*Expanded to*
bbc		chkbit; bno
bbs		chkbit; bo
cmpibe	cmpije	cmpi; be
cmpibg	cmpijg	cmpi; bg
cmpibge	cmpijge	cmpi; bge
cmpibl	cmpijl	cmpi; bl
cmpible	cmpijle	cmpi; ble
cmpibno	cmpijno	cmpi; bno
cmpibne	cmpijne	cmpi; bne
cmpibo	cmpijo	cmpi; bo
cmpobe	cmpoje	cmpo; be
cmpobg	cmpojg	cmpo; bg
cmpobge	cmpojge	cmpo; bge
cmpobl	cmpojl	cmpo; bl
cmpoble	cmpojle	cmpo; ble
cmpobne	cmpojne	cmpo; bne

8.8 M680x0 Dependent Features

8.8.1 M680x0 Options

The Motorola 680x0 version of `as` has two machine dependent options. One shortens undefined references from 32 to 16 bits, while the other is used to tell `as` what kind of machine it is assembling for.

You can use the '`-l`' option to shorten the size of references to undefined symbols. If you do not use the '`-l`' option, references to undefined symbols are wide enough for a full `long` (32 bits). (Since `as` cannot know where these symbols end up, `as` can only allocate space for the linker to fill in later. Since `as` does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only one word wide (16 bits). This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols are always less than 17 bits away.

The 680x0 version of `as` is most frequently used to assemble programs for the Motorola MC68020 microprocessor. Occasionally it is used to assemble programs for the mostly similar, but slightly different MC68000 or MC68010 microprocessors. You can give `as` the options '`-m68000`', '`-mc68000`', '`-m68010`', '`-mc68010`', '`-m68020`', and '`-mc68020`' to tell it what processor is the target.

8.8.2 Syntax

This syntax for the Motorola 680x0 was developed at MIT.

The 680x0 version of `as` uses syntax compatible with the Sun assembler. Intervening periods are ignored; for example, '`movl`' is equivalent to '`move.l`'.

In the following table *apc* stands for any of the address registers ('`a0`' through '`a7`'), nothing, (''), the Program Counter ('`pc`'), or the zero-address relative to the program counter ('`zpc`').

The following addressing modes are understood:

Immediate
: '`#`*`digits`*'

Data Register
: '`%d0`' through '`%d7`'

Address Register
: '`%a0`' through '`%a7`'
: '`%a7`' is also known as '`%sp`', i.e. the Stack Pointer. `%a6` is also known as '`%fp`', the Frame Pointer.

Address Register Indirect
: '%a0@' through '%a7@'

Address Register Postincrement
: '%a0@+' through '%a7@+'

Address Register Predecrement
: '%a0@-' through '%a7@-'

Indirect Plus Offset
: '%*apc*@(*digits*)'

Index
: '%*apc*@(*digits*,%*register*:*size*:*scale*)'
or '%*apc*@(%*register*:*size*:*scale*)'

Postindex
: '%*apc*@(*digits*)@(*digits*,%*register*:*size*:*scale*)'
or '%*apc*@(*digits*)@(%*register*:*size*:*scale*)'

Preindex
: '%*apc*@(*digits*,%*register*:*size*:*scale*)@(*digits*)'
or '%*apc*@(%*register*:*size*:*scale*)@(*digits*)'

Memory Indirect
: '%*apc*@(*digits*)@(*digits*)'

Absolute
: '*symbol*', or '*digits*'

For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a '%' before any use of a register name. This is intended to let the assembler distinguish between C variables and registers named 'a0' through 'a7', and so on. The '%' is always accepted, but is not required for certain configurations, notably 'sun3'.

8.8.3 Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax already discussed (see Section 8.8.2 "Syntax," page 79). as can accept some forms of Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction. The two kinds of syntax are fully compatible; our support for Motorola syntax is simply incomplete at present.

In particular, you may write or generate M68K assembler with the following conventions:

(In the following table *%apc* stands for any of the address registers ('%a0' through '%a7'), nothing ("), the Program Counter ('%pc'), or the zero-address relative to the program counter ('%zpc').)

The following additional addressing modes are understood:

Address Register Indirect
: '`%a0`' through '`%a7`'
'`%a7`' is also known as '`%sp`', i.e. the Stack Pointer. `%a6` is also known as '`%fp`', the Frame Pointer.

Address Register Postincrement
: '`(%a0)+`' through '`(%a7)+`'

Address Register Predecrement
: '`-(%a0)`' through '`-(%a7)`'

Indirect Plus Offset
: '`digits(%apc)`'

Index
: '`digits(%apc,(%register.size*scale))`'
or '`(%apc,%register.size*scale)`'
In either case, `size` and `scale` are optional (`scale` defaults to '`1`', `size` defaults to '`l`'). `scale` can be '`1`', '`2`', '`4`', or '`8`'. `size` can be '`w`' or '`l`'. `scale` is only supported on the 68020 and greater.

Other, more complex addressing modes permitted in Motorola syntax are not handled.

8.8.4 Floating Point

The floating point code is not too well tested, and may have subtle bugs in it.

Packed decimal (P) format floating literals are not supported. Feel free to add the code!

The floating point formats generated by directives are these.

`.float`
: `Single` precision floating point constants.

`.double`
: `Double` precision floating point constants.

There is no directive to produce regions of memory holding extended precision numbers, however they can be used as immediate operands to floating-point instructions. Adding a directive to create extended precision numbers would not be hard, but it has not yet seemed necessary.

8.8.5 680x0 Machine Directives

In order to be compatible with the Sun assembler the 680x0 assembler understands the following directives.

`.data1`
: This directive is identical to a `.data 1` directive.

`.data2` This directive is identical to a `.data 2` directive.

`.even` This directive is identical to a `.align 1` directive.

`.skip` This directive is identical to a `.space` directive.

8.8.6 Opcodes

8.8.6.1 Branch Improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that reach the target. Generally these mnemonics are made by substituting 'j' for 'b' at the start of a Motorola mnemonic.

The following table summarizes the pseudo-operations. A * flags cases that are more fully described after the table:

```
          Displacement
          +-------------------------------------------------
          |                68020   68000/10
Pseudo-Op |BYTE    WORD    LONG    LONG      non-PC relative
          +-------------------------------------------------
     jbsr |bsrs    bsr     bsrl    jsr       jsr
      jra |bras    bra     bral    jmp       jmp
*     jXX |bXXs    bXX     bXXl    bNXs;jmpl bNXs;jmp
*    dbXX |dbXX    dbXX        dbXX; bra; jmpl
*    fjXX |fbXXw   fbXXw   fbXXl             fbNXw;jmp

XX: condition
NX: negative of condition XX
```

*—see full description below

`jbsr`
`jra` These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target.

`j`*`XX`* Here, 'j*XX*' stands for an entire family of pseudo-operations, where *XX* is a conditional branch or condition-code test. The full list of pseudo-ops in this family is:

```
jhi   jls   jcc   jcs   jne   jeq   jvc
jvs   jpl   jmi   jge   jlt   jgt   jle
```

For the cases of non-PC relative displacements and long displacements on the 68000 or 68010, `as` issues a longer code fragment in terms of *NX*, the opposite condition to *XX*. For example, for the non-PC relative case:

```
     jXX foo
```

gives

```
          bNXs oof
          jmp foo
     oof:
```

`dbXX` The full family of pseudo-operations covered here is

```
dbhi   dbls   dbcc   dbcs   dbne   dbeq   dbvc
dbvs   dbpl   dbmi   dbge   dblt   dbgt   dble
dbf    dbra   dbt
```

Other than for word and byte displacements, when the source reads '`dbXX foo`', `as` emits

```
          dbXX oo1
          bra oo2
     oo1:jmpl foo
     oo2:
```

`fjXX` This family includes

```
fjne   fjeq   fjge   fjlt   fjgt   fjle   fjf
fjt    fjgl   fjgle  fjnge  fjngl  fjngle fjngt
fjnle  fjnlt  fjoge  fjogl  fjogt  fjole  fjolt
fjor   fjseq  fjsf   fjsne  fjst   fjueq  fjuge
fjugt  fjule  fjult  fjun
```

For branch targets that are not PC relative, `as` emits

```
          fbNX oof
          jmp foo
     oof:
```

when it encounters '`fjXX foo`'.

8.8.6.2 Special Characters

The immediate character is '`#`' for Sun compatibility. The line-comment character is '`|`'. If a '`#`' appears at the beginning of a line, it is treated as a comment unless it looks like '`# line file`', in which case it is treated normally.

8.9 SPARC Dependent Features

8.9.1 Options

The SPARC chip family includes several successive levels (or other variants) of chip, using the same core instruction set, but including a few additional instructions at each level.

By default, `as` assumes the core instruction set (SPARC v6), but "bumps" the architecture level as needed: it switches to successively higher architectures as it encounters instructions that only exist in the higher levels.

`-Av6 | -Av7 | -Av8 | -Av9 | -Asparclite`
: Use one of the '`-A`' options to select one of the SPARC architectures explicitly. If you select an architecture explicitly, `as` reports a fatal error if it encounters an instruction or feature requiring a higher level.

`-bump`
: Permit the assembler to "bump" the architecture level as required, but warn whenever it is necessary to switch to another level.

8.9.2 Floating Point

The Sparc uses IEEE floating-point numbers.

8.9.3 Sparc Machine Directives

The Sparc version of `as` supports the following additional machine directives:

`.align`
: This must be followed by the desired alignment in bytes.

`.common`
: This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.comm`, but the syntax is different.

`.half`
: This is functionally identical to `.short`.

`.proc`
: This directive is ignored. Any text following it on the same line is also ignored.

`.reserve`
: This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.lcomm`, but the syntax is different.

`.seg` This must be followed by `"text"`, `"data"`, or `"data1"`. It behaves like `.text`, `.data`, or `.data 1`.

`.skip` This is functionally identical to the `.space` directive.

`.word` On the Sparc, the `.word` directive produces 32 bit values, instead of the 16 bit values it produces on many other machines.

`.xword` On the Sparc V9 processor, the `.xword` directive produces 64 bit values.

8.10 80386 Dependent Features

8.10.1 Options

The 80386 has no machine dependent options.

8.10.2 AT&T Syntax versus Intel Syntax

In order to maintain compatibility with the output of `gcc`, `as` supports AT&T System V/386 assembler syntax. This is quite different from Intel syntax. We mention these differences because almost all 80386 documents used only Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by '`$`'; Intel immediate operands are undelimited (Intel '`push 4`' is AT&T '`pushl $4`'). AT&T register operands are preceded by '`%`'; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by '`*`'; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel '`add eax, 4`' is '`addl $4, %eax`'. The '`source, dest`' convention is maintained for compatibility with previous Unix assemblers.
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of '`b`', '`w`', and '`l`' specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixes memory operands (*not* the opcodes themselves) with '`byte ptr`', '`word ptr`', and '`dword ptr`'. Thus, Intel '`mov al, byte ptr` *foo*' is '`movb` *foo*`, %al`' in AT&T syntax.
- Immediate form long jumps and calls are '`lcall/ljmp $`*section*`, $`*offset*' in AT&T syntax; the Intel syntax is '`call/jmp far` *section*`:`*offset*'. Also, the far return instruction is '`lret $`*stack-adjust*' in AT&T syntax; Intel syntax is '`ret far` *stack-adjust*'.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

8.10.3 Opcode Naming

Opcode names are suffixed with one character modifiers which specify the size of operands. The letters '`b`', '`w`', and '`l`' specify byte, word, and long operands. If no suffix is specified by an instruction and it contains

no memory operands then `as` tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, '`mov %ax, %bx`' is equivalent to '`movw %ax, %bx`'; also, '`mov $1, %bx`' is equivalent to '`movw $1, %bx`'. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing opcode suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the opcode suffix.)

Almost all opcodes have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend *from* and a size to zero extend *to*. This is accomplished by using two opcode suffixes in AT&T syntax. Base names for sign extend and zero extend are '`movs...`' and '`movz...`' in AT&T syntax ('`movsx`' and '`movzx`' in Intel syntax). The opcode suffixes are tacked on to this base name, the *from* suffix before the *to* suffix. Thus, '`movsbl %al, %edx`' is AT&T syntax for "move sign extend *from* %al *to* %edx." Possible suffixes, thus, are '`bl`' (from byte to long), '`bw`' (from byte to word), and '`wl`' (from word to long).

The Intel-syntax conversion instructions

- '`cbw`' — sign-extend byte in '`%al`' to word in '`%ax`',
- '`cwde`' — sign-extend word in '`%ax`' to long in '`%eax`',
- '`cwd`' — sign-extend word in '`%ax`' to long in '`%dx:%ax`',
- '`cdq`' — sign-extend dword in '`%eax`' to quad in '`%edx:%eax`',

are called '`cbtw`', '`cwtl`', '`cwtd`', and '`cltd`' in AT&T naming. `as` accepts either naming for these instructions.

Far call/jump instructions are '`lcall`' and '`ljmp`' in AT&T syntax, but are '`call far`' and '`jump far`' in Intel convention.

8.10.4 Register Naming

Register operands are always prefixes with '`%`'. The 80386 registers consist of

- the 8 32-bit registers '`%eax`' (the accumulator), '`%ebx`', '`%ecx`', '`%edx`', '`%edi`', '`%esi`', '`%ebp`' (the frame pointer), and '`%esp`' (the stack pointer).
- the 8 16-bit low-ends of these: '`%ax`', '`%bx`', '`%cx`', '`%dx`', '`%di`', '`%si`', '`%bp`', and '`%sp`'.
- the 8 8-bit registers: '`%ah`', '`%al`', '`%bh`', '`%bl`', '`%ch`', '`%cl`', '`%dh`', and '`%dl`' (These are the high-bytes and low-bytes of '`%ax`', '`%bx`', '`%cx`', and '`%dx`')
- the 6 section registers '`%cs`' (code section), '`%ds`' (data section), '`%ss`' (stack section), '`%es`', '`%fs`', and '`%gs`'.

- the 3 processor control registers '%cr0', '%cr2', and '%cr3'.
- the 6 debug registers '%db0', '%db1', '%db2', '%db3', '%db6', and '%db7'.
- the 2 test registers '%tr6' and '%tr7'.
- the 8 floating point register stack '%st' or equivalently '%st(0)', '%st(1)', '%st(2)', '%st(3)', '%st(4)', '%st(5)', '%st(6)', and '%st(7)'.

8.10.5 Opcode Prefixes

Opcode prefixes are used to modify the following opcode. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to give operand and address size (16-bit operands are specified in an instruction by prefixing what would normally be 32-bit operands with a "operand size" opcode prefix). Opcode prefixes are usually given as single-line instructions with no operands, and must directly precede the instruction they act upon. For example, the 'scas' (scan string) instruction is repeated with:

```
repne
scas
```

Here is a list of opcode prefixes:

- Section override prefixes 'cs', 'ds', 'ss', 'es', 'fs', 'gs'. These are automatically added by specifying using the *section*:*memory-operand* form for memory references.
- Operand/Address size prefixes 'data16' and 'addr16' change 32-bit operands/addresses into 16-bit operands/addresses. Note that 16-bit addressing modes (i.e. 8086 and 80286 addressing modes) are not supported (yet).
- The bus lock prefix 'lock' inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).
- The wait for coprocessor prefix 'wait' waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.
- The 'rep', 'repe', and 'repne' prefixes are added to string instructions to make them repeat '%ecx' times.

8.10.6 Memory References

An Intel syntax indirect memory reference of the form

section:[*base* + *index***scale* + *disp*]

is translated into the AT&T syntax

section:*disp*(*base*, *index*, *scale*)

where *base* and *index* are the optional 32-bit base and index registers, *disp* is the optional displacement, and *scale*, taking the values 1, 2, 4, and 8, multiplies *index* to calculate the address of the operand. If no *scale* is specified, *scale* is taken to be 1. *section* specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in AT&T syntax *must* have be preceded by a '%'. If you specify a section override which coincides with the default section register, as does *not* output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

AT&T: '-4(%ebp)', Intel: '[ebp - 4]'
: *base* is '%ebp'; *disp* is '-4'. *section* is missing, and the default section is used ('%ss' for addressing with '%ebp' as the base register). *index*, *scale* are both missing.

AT&T: 'foo(,%eax,4)', Intel: '[foo + eax*4]'
: *index* is '%eax' (scaled by a *scale* 4); *disp* is 'foo'. All other fields are missing. The section register here defaults to '%ds'.

AT&T: 'foo(,1)'; Intel '[foo]'
: This uses the value pointed to by 'foo' as a memory operand. Note that *base* and *index* are both missing, but there is only *one* ','. This is a syntactic exception.

AT&T: '%gs:foo'; Intel 'gs:foo'
: This selects the contents of the variable 'foo' with section register *section* being '%gs'.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with '*'. If no '*' is specified, as always chooses PC relative addressing for jump/call labels.

Any instruction that has a memory operand *must* specify its size (byte, word, or long) with an opcode suffix ('b', 'w', or 'l', respectively).

8.10.7 Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e. prefixing the jump instruction with the 'addr16' opcode prefix), since the 80386 insists upon masking '%eip' to 16 bits after the word displacement is added.

Note that the 'jcxz', 'jecxz', 'loop', 'loopz', 'loope', 'loopnz' and 'loopne' instructions only come in byte displacements, so that if you use these instructions (gcc does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding 'jcxz foo' to

```
          jcxz cx_zero
          jmp cx_nonzero
cx_zero:  jmp foo
cx_nonzero:
```

8.10.8 Floating Point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an opcode suffix and a constructor associated with it. Opcode suffixes specify operand's data types. Constructors build these data types into memory.

- Floating point constructors are '.float' or '.single', '.double', and '.tfloat' for 32-, 64-, and 80-bit formats. These correspond to opcode suffixes 's', 'l', and 't'. 't' stands for temporary real, and that the 80387 only supports this format via the 'fldt' (load temporary real to stack top) and 'fstpt' (store temporary real and pop stack) instructions.
- Integer constructors are '.word', '.long' or '.int', and '.quad' for the 16-, 32-, and 64-bit integer formats. The corresponding opcode suffixes are 's' (single), 'l' (long), and 'q' (quad). As with the temporary real format the 64-bit 'q' format is only present in the 'fildq' (load quad integer to stack top) and 'fistpq' (store quad integer and pop stack) instructions.

Register to register operations do not require opcode suffixes, so that 'fst %st, %st(1)' is equivalent to 'fstl %st, %st(1)'.

Since the 80387 automatically synchronizes with the 80386 'fwait' instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, as suppresses the 'fwait' instruction whenever it is implicitly selected by one of the 'fn. . .' instructions. For example, 'fsave' and 'fnsave' are treated identically. In general, all the 'fn. . .' instructions are made equivalent to 'f. . .' instructions. If 'fwait' is desired it must be explicitly coded.

8.10.9 Writing 16-bit Code

While GAS normally writes only "pure" 32-bit i386 code, it has limited support for writing code to run in real mode or in 16-bit protected mode code segments. To do this, insert a '`.code16`' directive before the assembly language instructions to be run in 16-bit mode. You can switch GAS back to writing normal 32-bit code with the '`.code32`' directive.

GAS understands exactly the same assembly language syntax in 16-bit mode as in 32-bit mode. The function of any given instruction is exactly the same regardless of mode, as long as the resulting object code is executed in the mode for which GAS wrote it. So, for example, the '`ret`' mnemonic produces a 32-bit return instruction regardless of whether it is to be run in 16-bit or 32-bit mode. (If GAS is in 16-bit mode, it will add an operand size prefix to the instruction to force it to be a 32-bit return.)

This means, for one thing, that you can use GNU CC to write code to be run in real mode or 16-bit protected mode. Just insert the statement '`asm(".code16");`' at the beginning of your C source file, and while GNU CC will still be generating 32-bit code, GAS will automatically add all the necessary size prefixes to make that code run in 16-bit mode. Of course, since GNU CC only writes small-model code (it doesn't know how to attach segment selectors to pointers like native x86 compilers do), any 16-bit code you write with GNU CC will essentially be limited to a 64K address space. Also, there will be a code size and performance penalty due to all the extra address and operand size prefixes GAS has to add to the instructions.

Note that placing GAS in 16-bit mode does not mean that the resulting code will necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you would have to refrain from using *any* 32-bit constructs which require GAS to output address or operand size prefixes. At the moment this would be rather difficult, because GAS currently supports *only* 32-bit addressing modes: when writing 16-bit code, it *always* outputs address size prefixes for any instruction that uses a non-register addressing mode. So you can write code that runs on 16-bit processors, but only if that code never references memory.

8.10.10 Notes

There is some trickery concerning the '`mul`' and '`imul`' instructions that deserves mention. The 16-, 32-, and 64-bit expanding multiplies (base opcode '`0xf6`'; extension 4 for '`mul`' and 5 for '`imul`') can be output only in the one operand form. Thus, '`imul %ebx, %eax`' does *not* select the expanding multiply; the expanding multiply would clobber the '`%edx`' register, and this would confuse `gcc` output. Use '`imul %ebx`' to get the 64-bit product in '`%edx:%eax`'.

We have added a two operand form of 'imul' when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying '%eax' by 69, for example, can be done with 'imul $69, %eax' rather than 'imul $69, %eax, %eax'.

8.11 Z8000 Dependent Features

The Z8000 as supports both members of the Z8000 family: the unsegmented Z8002, with 16 bit addresses, and the segmented Z8001 with 24 bit addresses.

When the assembler is in unsegmented mode (specified with the `unsegm` directive), an address takes up one word (16 bit) sized register. When the assembler is in segmented mode (specified with the `segm` directive), a 24-bit address takes up a long (32 bit) register. See Section 8.11.3 "Assembler Directives for the Z8000," page 94, for a list of other Z8000 specific assembler directives.

8.11.1 Options

`as` has no additional command-line options for the Zilog Z8000 family.

8.11.2 Syntax

8.11.2.1 Special Characters

'`!`' is the line comment character.

You can use '`;`' instead of a newline to separate statements.

8.11.2.2 Register Names

The Z8000 has sixteen 16 bit registers, numbered 0 to 15. You can refer to different sized groups of registers by register number, with the prefix '`r`' for 16 bit registers, '`rr`' for 32 bit registers and '`rq`' for 64 bit registers. You can also refer to the contents of the first eight (of the sixteen 16 bit registers) by bytes. They are named '`r`*n*`h`' and '`r`*n*`l`'.

byte registers

```
r0l r0h r1h r1l r2h r2l r3h r3l
r4h r4l r5h r5l r6h r6l r7h r7l
```

word registers

```
r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
```

long word registers

```
rr0 rr2 rr4 rr6 rr8 rr10 rr12 rr14
```

quad word registers

```
rq0 rq4 rq8 rq12
```

8.11.2.3 Addressing Modes

as understands the following addressing modes for the Z8000:

`rn` Register direct

`@rn` Indirect register

addr Direct: the 16 bit or 24 bit address (depending on whether the assembler is in segmented or unsegmented mode) of the operand is in the instruction.

`address(rn)`
Indexed: the 16 or 24 bit address is added to the 16 bit register to produce the final address in memory of the operand.

`rn(#imm)` Base Address: the 16 or 24 bit register is added to the 16 bit sign extended immediate displacement to produce the final address in memory of the operand.

`rn(rm)` Base Index: the 16 or 24 bit register r*n* is added to the sign extended 16 bit index register r*m* to produce the final address in memory of the operand.

`#xx` Immediate data *xx*.

8.11.3 Assembler Directives for the Z8000

The Z8000 port of as includes these additional assembler directives, for compatibility with other Z8000 assemblers. As shown, these do not begin with '.' (unlike the ordinary as directives).

`segm` Generates code for the segmented Z8001.

`unsegm` Generates code for the unsegmented Z8002.

`name` Synonym for `.file`

`global` Synonum for `.global`

`wval` Synonym for `.word`

`lval` Synonym for `.long`

`bval` Synonym for `.byte`

`sval` Assemble a string. `sval` expects one string literal, delimited by single quotes. It assembles each byte of the string into consecutive addresses. You can use the escape sequence '`%xx`' (where *xx* represents a two-digit hexadecimal number) to represent the character whose ASCII value is *xx*. Use this feature to describe single quote and other characters that

may not appear in string literals as themselves. For example, the C statement '`char *a = "he said \"it's 50% off\"";`' is represented in Z8000 assembly language (shown with the assembler output in hex at the left) as

```
68652073    sval    'he said %22it%27s 50%25 off%22%00'
61696420
22697427
73203530
25206F66
662200
```

`rsect` synonym for `.section`

`block` synonym for `.space`

`even` synonym for `.align 1`

8.11.4 Opcodes

For detailed information on the Z8000 machine instruction set, see *Z8000 Technical Manual*.

The following table summarizes the opcodes and their arguments:

`rs`	16 bit source register
`rd`	16 bit destination register
`rbs`	8 bit source register
`rbd`	8 bit destination register
`rrs`	32 bit source register
`rrd`	32 bit destination register
`rqs`	64 bit source register
`rqd`	64 bit destination register
`addr`	16/24 bit address
`imm`	immediate data

```
adc rd,rs
adcb rbd,rbs
add rd,@rs
add rd,addr
add rd,addr(rs)
add rd,imm16
add rd,rs
addb rbd,@rs
addb rbd,addr
addb rbd,addr(rs)
addb rbd,imm8
addb rbd,rbs
addl rrd,@rs
addl rrd,addr
addl rrd,addr(rs)
addl rrd,imm32
addl rrd,rrs
clrb addr
clrb addr(rd)
clrb rbd
com @rd
com addr
com addr(rd)
com rd
comb @rd
comb addr
comb addr(rd)
comb rbd
comflg flags
cp @rd,imm16
cp addr(rd),imm16
cp addr,imm16
cp rd,@rs
cp rd,addr
cpsir @rd,@rs,rr,cc
cpsirb @rd,@rs,rr,cc
dab rbd
dbjnz rbd,disp7
dec @rd,imm4m1
dec addr(rd),imm4m1
dec addr,imm4m1
dec rd,imm4m1
decb @rd,imm4m1
decb addr(rd),imm4m1
decb addr,imm4m1
decb rbd,imm4m1
di i2
div rrd,@rs
div rrd,addr
div rrd,addr(rs)
div rrd,imm16
```

```
and rd,@rs
and rd,addr
and rd,addr(rs)
and rd,imm16
and rd,rs
andb rbd,@rs
andb rbd,addr
andb rbd,addr(rs)
andb rbd,imm8
andb rbd,rbs
bit @rd,imm4
bit addr(rd),imm4
bit addr,imm4
bit rd,imm4
bit rd,rs
bitb @rd,imm4
bitb addr(rd),imm4
bitb addr,imm4
bitb rbd,imm4
bitb rbd,rs
bpt
call @rd
call addr
call addr(rd)
calr disp12
clr @rd
clr addr
clr addr(rd)
clr rd
clrb @rd
inc addr,imm4m1
inc rd,imm4m1
incb @rd,imm4m1
incb addr(rd),imm4m1
incb addr,imm4m1
incb rbd,imm4m1
ind @rd,@rs,ra
indb @rd,@rs,rba
inib @rd,@rs,ra
inibr @rd,@rs,ra
iret
jp cc,@rd
jp cc,addr
jp cc,addr(rd)
jr cc,disp8
ld @rd,imm16
ld @rd,rs
ld addr(rd),imm16
ld addr(rd),rs
ld addr,imm16
ld addr,rs
ld rd(imm16),rs
ld rd(rx),rs
cp rd,addr(rs)
cp rd,imm16
cp rd,rs
cpb @rd,imm8
cpb addr(rd),imm8
cpb addr,imm8
cpb rbd,@rs
cpb rbd,addr
cpb rbd,addr(rs)
cpb rbd,imm8
cpb rbd,rbs
cpd rd,@rs,rr,cc
cpdb rbd,@rs,rr,cc
cpdr rd,@rs,rr,cc
cpdrb rbd,@rs,rr,cc
cpi rd,@rs,rr,cc
cpib rbd,@rs,rr,cc
cpir rd,@rs,rr,cc
cpirb rbd,@rs,rr,cc
cpl rrd,@rs
cpl rrd,addr
cpl rrd,addr(rs)
cpl rrd,imm32
cpl rrd,rrs
cpsd @rd,@rs,rr,cc
cpsdb @rd,@rs,rr,cc
cpsdr @rd,@rs,rr,cc
cpsdrb @rd,@rs,rr,cc
cpsi @rd,@rs,rr,cc
cpsib @rd,@rs,rr,cc
ldb rbd,rs(rx)
ldb rd(imm16),rbs
ldb rd(rx),rbs
ldctl ctrl,rs
ldctl rd,ctrl
ldd @rs,@rd,rr
lddb @rs,@rd,rr
lddr @rs,@rd,rr
lddrb @rs,@rd,rr
ldi @rd,@rs,rr
ldib @rd,@rs,rr
ldir @rd,@rs,rr
ldirb @rd,@rs,rr
ldk rd,imm4
ldl @rd,rrs
ldl addr(rd),rrs
ldl addr,rrs
ldl rd(imm16),rrs
ldl rd(rx),rrs
ldl rrd,@rs
ldl rrd,addr
ldl rrd,addr(rs)
ldl rrd,imm32
div rrd,rs
divl rqd,@rs
divl rqd,addr
divl rqd,addr(rs)
divl rqd,imm32
divl rqd,rrs
djnz rd,disp7
ei i2
ex rd,@rs
ex rd,addr
ex rd,addr(rs)
ex rd,rs
exb rbd,@rs
exb rbd,addr
exb rbd,addr(rs)
exb rbd,rbs
ext0e imm8
ext0f imm8
ext8e imm8
ext8f imm8
exts rrd
extsb rd
extsl rqd
halt
in rd,@rs
in rd,imm16
inb rbd,@rs
inb rbd,imm16
inc @rd,imm4m1
inc addr(rd),imm4m1
mult rrd,addr(rs)
mult rrd,imm16
mult rrd,rs
multl rqd,@rs
multl rqd,addr
multl rqd,addr(rs)
multl rqd,imm32
multl rqd,rrs
neg @rd
neg addr
neg addr(rd)
neg rd
negb @rd
negb addr
negb addr(rd)
negb rbd
nop
or rd,@rs
or rd,addr
or rd,addr(rs)
or rd,imm16
or rd,rs
orb rbd,@rs
```

```
ld rd,@rs
ld rd,addr
ld rd,addr(rs)
ld rd,imm16
ld rd,rs
ld rd,rs(imm16)
ld rd,rs(rx)
lda rd,addr
lda rd,addr(rs)
lda rd,rs(imm16)
lda rd,rs(rx)
ldar rd,disp16
ldb @rd,imm8
ldb @rd,rbs
ldb addr(rd),imm8
ldb addr(rd),rbs
ldb addr,imm8
ldb addr,rbs
ldb rbd,@rs
ldb rbd,addr
ldb rbd,addr(rs)
ldb rbd,imm8
ldb rbd,rbs
ldb rbd,rs(imm16)
push @rd,rs
pushl @rd,@rs
pushl @rd,addr
pushl @rd,addr(rs)
pushl @rd,rrs
res @rd,imm4
res addr(rd),imm4
res addr,imm4
res rd,imm4
res rd,rs
resb @rd,imm4
resb addr(rd),imm4
resb addr,imm4
resb rbd,imm4
resb rbd,rs
resflg imm4
ret cc
rl rd,imm1or2
rlb rbd,imm1or2
rlc rd,imm1or2
rlcb rbd,imm1or2
rldb rbb,rba
rr rd,imm1or2
rrb rbd,imm1or2
rrc rd,imm1or2
rrcb rbd,imm1or2
rrdb rbb,rba
rsvd36
rsvd38
ldl rrd,rrs
ldl rrd,rs(imm16)
ldl rrd,rs(rx)
ldm @rd,rs,n
ldm addr(rd),rs,n
ldm addr,rs,n
ldm rd,@rs,n
ldm rd,addr(rs),n
ldm rd,addr,n
ldps @rs
ldps addr
ldps addr(rs)
ldr disp16,rs
ldr rd,disp16
ldrb disp16,rbs
ldrb rbd,disp16
ldrl disp16,rrs
ldrl rrd,disp16
mbit
mreq rd
mres
mset
mult rrd,@rs
mult rrd,addr
set addr,imm4
set rd,imm4
set rd,rs
setb @rd,imm4
setb addr(rd),imm4
setb addr,imm4
setb rbd,imm4
setb rbd,rs
setflg imm4
sinb rbd,imm16
sinb rd,imm16
sind @rd,@rs,ra
sindb @rd,@rs,rba
sinib @rd,@rs,ra
sinibr @rd,@rs,ra
sla rd,imm8
slab rbd,imm8
slal rrd,imm8
sll rd,imm8
sllb rbd,imm8
slll rrd,imm8
sout imm16,rs
soutb imm16,rbs
soutd @rd,@rs,ra
soutdb @rd,@rs,rba
soutib @rd,@rs,ra
soutibr @rd,@rs,ra
sra rd,imm8
srab rbd,imm8
orb rbd,addr
orb rbd,addr(rs)
orb rbd,imm8
orb rbd,rbs
out @rd,rs
out imm16,rs
outb @rd,rbs
outb imm16,rbs
outd @rd,@rs,ra
outdb @rd,@rs,rba
outib @rd,@rs,ra
outibr @rd,@rs,ra
pop @rd,@rs
pop addr(rd),@rs
pop addr,@rs
pop rd,@rs
popl @rd,@rs
popl addr(rd),@rs
popl addr,@rs
popl rrd,@rs
push @rd,@rs
push @rd,addr
push @rd,addr(rs)
push @rd,imm16
subl rrd,imm32
subl rrd,rrs
tcc cc,rd
tccb cc,rbd
test @rd
test addr
test addr(rd)
test rd
testb @rd
testb addr
testb addr(rd)
testb rbd
testl @rd
testl addr
testl addr(rd)
testl rrd
trdb @rd,@rs,rba
trdrb @rd,@rs,rba
trib @rd,@rs,rbr
trirb @rd,@rs,rbr
trtdrb @ra,@rb,rbr
trtib @ra,@rb,rr
trtirb @ra,@rb,rbr
trtrb @ra,@rb,rbr
tset @rd
tset addr
tset addr(rd)
tset rd
tsetb @rd
```

```
rsvd78
rsvd7e
rsvd9d
rsvd9f
rsvdb9
rsvdbf
sbc rd,rs
sbcb rbd,rbs
sc imm8
sda rd,rs
sdab rbd,rs
sdal rrd,rs
sdl rd,rs
sdlb rbd,rs
sdll rrd,rs
set @rd,imm4
set addr(rd),imm4
sral rrd,imm8
srl rd,imm8
srlb rbd,imm8
srll rrd,imm8
sub rd,@rs
sub rd,addr
sub rd,addr(rs)
sub rd,imm16
sub rd,rs
subb rbd,@rs
subb rbd,addr
subb rbd,addr(rs)
subb rbd,imm8
subb rbd,rbs
subl rrd,@rs
subl rrd,addr
subl rrd,addr(rs)
tsetb addr
tsetb addr(rd)
tsetb rbd
xor rd,@rs
xor rd,addr
xor rd,addr(rs)
xor rd,imm16
xor rd,rs
xorb rbd,@rs
xorb rbd,addr
xorb rbd,addr(rs)
xorb rbd,imm8
xorb rbd,rbs
xorb rbd,rbs
```

8.12 MIPS Dependent Features

GNU as for MIPS architectures supports the MIPS R2000, R3000, R4000 and R6000 processors. For information about the MIPS instruction set, see *MIPS RISC Architecture*, by Kane and Heindrich (Prentice-Hall). For an overview of MIPS assembly conventions, see "Appendix D: Assembly Language Programming" in the same work.

8.12.1 Assembler options

The MIPS configurations of GNU as support these special options:

-G *num*
: This option sets the largest size of an object that can be referenced implicitly with the gp register. It is only accepted for targets that use ECOFF format. The default value is 8.

-EB
-EL
: Any MIPS configuration of as can select big-endian or little-endian output at run time (unlike the other GNU development tools, which must be configured for one or the other). Use '-EB' to select big-endian output, and '-EL' for little-endian.

-mips1
-mips2
-mips3
: Generate code for a particular MIPS Instruction Set Architecture level. '-mips1' corresponds to the R2000 and R3000 processors, '-mips2' to the R6000 processor, and '-mips3' to the R4000 processor. You can also switch instruction sets during the assembly; see Section 8.12.4 "Directives to override the ISA level," page 101.

-m4650

-no-m4650
: Generate code for the MIPS R4650 chip. This tells the assembler to accept the 'mad' and 'madu' instruction, and to not schedule 'nop' instructions around accesses to the 'HI' and 'LO' registers. '-no-m4650' turns off this option.

-mcpu=*CPU*
: Generate code for a particular MIPS cpu. This has little effect on the assembler, but it is passed by gcc.

-nocpp
: This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With GNU as, there is no need for '-nocpp', because the GNU assembler itself never runs the C preprocessor.

`--trap`
`--no-break`
: as automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes as to generate code to take a trap exception rather than a break exception when an error is detected. The trap instructions are only supported at Instruction Set Architecture level 2 and higher.

`--break`
`--no-trap`
: Generate code to take a break exception rather than a trap exception when an error is detected. This is the default.

8.12.2 MIPS ECOFF object code

Assembling for a MIPS ECOFF target supports some additional sections besides the usual `.text`, `.data` and `.bss`. The additional sections are `.rdata`, used for read-only data, `.sdata`, used for small data, and `.sbss`, used for small common objects.

When assembling for ECOFF, the assembler uses the `$gp` (`$28`) register to form the address of a "small object". Any object in the `.sdata` or `.sbss` sections is considered "small" in this sense. For external objects, or for objects in the `.bss` section, you can use the gcc '`-G`' option to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller uses `$gp`. Passing '`-G 0`' to as prevents it from using the `$gp` register on the basis of object size (but the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case). The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it. The size of an external object may be set with the `.extern` directive. For example, '`.extern sym,4`' declares that the object at `sym` is 4 bytes in length, whie leaving `sym` otherwise undefined.

Using small ECOFF objects requires linker support, and assumes that the `$gp` register is correctly initialized (normally done automatically by the startup code). MIPS ECOFF assembly code must not modify the `$gp` register.

8.12.3 Directives for debugging information

MIPS ECOFF as supports several directives used for generating debugging information which are not support by traditional MIPS assemblers. These are `.def`, `.endef`, `.dim`, `.file`, `.scl`, `.size`, `.tag`, `.type`, `.val`, `.stabd`, `.stabn`, and `.stabs`. The debugging information generated by the three `.stab` directives can only be read by GDB, not by traditional

MIPS debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers!

8.12.4 Directives to override the ISA level

GNU `as` supports an additional directive to change the MIPS Instruction Set Architecture level on the fly: `.set mipsn`. *n* should be a number from 0 to 3. A value from 1 to 3 makes the assembler accept instructions for the corresponding ISA level, from that point on in the assembly. `.set mipsn` affects not only which instructions are permitted, but also how certain macros are expanded. `.set mips0` restores the ISA level to its original level: either the level you selected with command line options, or the default for your configuration. You can use this feature to permit specific R4000 instructions while assembling in 32 bit mode. Use this directive with care!

Traditional MIPS assemblers do not support this directive.

9 Acknowledgements

If you have contributed to `as` and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the maintainer, and we'll correct the situation. Currently the maintainer is Ken Raeburn (email address `raeburn@cygnus.com`).

Dean Elsner wrote the original GNU assembler for the VAX.[1]

Jay Fenlason maintained GAS for a while, adding support for GDB-specific debug information and the 68k series machines, most of the preprocessing pass, and extensive changes in '`messages.c`', '`input-file.c`', '`write.c`'.

K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file format back ends (including heavy rewrite, testing, an integration of the coff and b.out back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ANSI C including full prototypes, added support for m680[34]0 and cpu32, did considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, rs6000, and hp300hpux host ports, updated "know" assertions and made them work, much other reorganization, cleanup, and lint.

Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.

The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

The Intel 80386 machine description was written by Eliot Dresselhaus.

Minh Tran-Le at IntelliCorp contributed some AIX 386 support.

The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.

Keith Knowles at the Open Software Foundation wrote the original MIPS back end ('`tc-mips.c`', '`tc-mips.h`'), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support a.out format.

Support for the Zilog Z8k and Hitachi H8/300 and H8/500 processors (tc-z8k, tc-h8300, tc-h8500), and IEEE 695 object file format (obj-ieee),

[1] Any more details?

was written by Steve Chamberlain of Cygnus Support. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.

John Gilmore built the AMD 29000 support, added `.include` support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g. `jsr`), while synthetic instructions remained shrinkable (`jbsr`). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS ECOFF and ELF targets, and made a few other minor patches.

Steve Chamberlain made `as` able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus Support.

Support for ELF format files has been worked on by Mark Eichin of Cygnus Support (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn of Cygnus Support (sparc, and some initial 64-bit support).

Several engineers at Cygnus Support have also provided many small bug fixes and configuration enhancements.

Many others have contributed large or small bugfixes and enhancements. If you have contributed significant work and are not mentioned on this list, and want to be, let us know. Some of the history has been lost; we are not intentionally leaving anyone out.

Index

#

..... 14
#APP 13
#NO_APP 13

$

$ in symbol names 62, 71

-

-- 5
--statistics 11
-+ option, VAX/VMS 50
-a 9
-A options, i960 75
-ad 9
-ah 9
-al 9
-an 9
-as 9
-Asparclite 84
-Av6 84
-Av8 84
-Av9 84
-b option, i960 75
-D 9
-D, ignored on VAX 49
-d, VAX option 49
-EB option (MIPS) 99
-EL option (MIPS) 99
-f 10
-G option (MIPS) 99
-h option, VAX/VMS 50
-I *path* 10
-J, ignored on VAX 49
-K 10
-L 10
-l option, M680x0 79
-m68000 and related options 79
-no-relax option, i960 76
-nocpp ignored (MIPS) 99
-o 11
-R 11
-S, ignored on VAX 49
-t, ignored on VAX 50
-T, ignored on VAX 49
-v 11
-V, redundant on VAX 49
-version 11
-W 11

.

. (symbol) 28
.o 6
.param on HPPA 67
.set mips*n* 101

:

: (label) 15

\

\" (doublequote character) 17
\\ ('\' character) 16
\b (backspace character) 16
ddd (octal character code) 16
\f (formfeed character) 16
\n (newline character) 16
\r (carriage return character) 16
\t (tab) 16
xdd (hex character code) 16

1

16-bit code, i386 91

2

29K support 54

A

a.out 6
a.out symbol attributes 29
abort directive 35
ABORT directive 35
absolute section 23
addition, permitted arguments 32
addresses 31

addresses, format of 22
addressing modes, H8/300 56
addressing modes, H8/500 63
addressing modes, M680x0 . . . . 79, 80
addressing modes, SH 71
addressing modes, Z8000 94
advancing location counter 42
`align` directive 35
`align` directive, SPARC 84
altered difference tables 47
alternate syntax for the 680x0 80
AMD 29K floating point (ieee) 55
AMD 29K identifiers 54
AMD 29K line comment character 54
AMD 29K line separator 54
AMD 29K machine directives 55
AMD 29K opcodes 55
AMD 29K options (none) 54
AMD 29K protected registers 54
AMD 29K register names 54
AMD 29K special purpose registers . . . 54
AMD 29K statement separator 54
AMD 29K support 54
`app-file` directive 36
architecture options, i960 75
architecture options, M680x0 79
architectures, SPARC 84
arguments for addition 32
arguments for subtraction 32
arguments in expressions 31
arithmetic functions 32
arithmetic operands 31
`as` version 11
`ascii` directive 36
`asciz` directive 36
assembler internal logic error 24
assembler, and linker 21
assembly listings, enabling 9
assigning values to symbols 27, 38
attributes, symbol 29
auxiliary attributes, COFF symbols . . . 30
auxiliary symbol information, COFF . . 37
`Av7` 84

B

backslash (`\\`) 16
backspace (`\b`) 16
big endian output, MIPS 3
big-endian output, MIPS 99
bignums 18
binary integers 17
bitfields, not supported on VAX 53
`block` 95
`block` directive, AMD 29K 55
branch improvement, M680x0 82
branch improvement, VAX 51
branch recording, i960 75
branch statistics table, i960 75
`bss` directive, i960 76
bss section 23, 25
bus lock prefixes, i386 88
`bval` 94
`byte` directive 36

C

call instructions, i386 87
`callj`, i960 pseudo-opcode 77
carriage return (`\r`) 16
character constants 16
character escape codes 16
character, single 17
characters used in symbols 14
`code16` directive, i386 91
`code32` directive, i386 91
COFF auxiliary symbol information . . 37
COFF named section 44
COFF structure debugging 46
COFF symbol attributes 30
COFF symbol descriptor 37
COFF symbol storage class 43
COFF symbol type 47
COFF symbols, debugging 37
COFF value attribute 47
`comm` directive 36
command line conventions 5
command-line options ignored, VAX . . 49
comments 14
comments, M680x0 83
comments, removed by preprocessor . . 13
`common` directive, SPARC 84
common variable storage 25
compare and jump expansions, i960 . . 78
compare/branch instructions, i960 78
conditional assembly 40
constant, single character 17
constants 16

constants, bignum 18
constants, character 16
constants, converted by preprocessor .. 13
constants, floating point 18
constants, integer 17
constants, number 17
constants, string 16
continuing statements 15
conversion instructions, i386 87
coprocessor wait, i386 88
`cputype` directive, AMD 29K 55
current address 28
current address, advancing 42

D

data and text sections, joining 11
`data` directive 36
data section 23
`data1` directive, M680x0 81
`data2` directive, M680x0 81
debuggers, and symbol order 27
debugging COFF symbols 37
decimal integers 17
`def` directive 37
deprecated directives 48
`desc` directive 37
descriptor, of `a.out` symbol 29
`dfloat` directive, VAX 50
difference tables altered 47
difference tables, warning 10
`dim` directive 37
directives and instructions 15
directives, M680x0 81
directives, machine independent 35
directives, Z8000 94
displacement sizing character, VAX ... 53
dot (symbol) 28
`double` directive 37
`double` directive, i386 90
`double` directive, M680x0 81
`double` directive, VAX 50
doublequote (`\"`) 17

E

ECOFF sections 100
eight-byte integer 43
`eject` directive 37
`else` directive 37
empty expressions 31
`endef` directive 38
endianness, MIPS 3
`endif` directive 38
EOF, newline must precede 15
`equ` directive 38
error messsages 6
errors, continuing after 12
escape codes, character 16
`even` 95
`even` directive, M680x0 82
expr (internal section) 24
expression arguments 31
expressions 31
expressions, empty 31
expressions, integer 31
`extended` directive, i96 76
`extern` directive 38

F

faster processing (`-f`) 10
`ffloat` directive, VAX 50
`file` directive 38
`file` directive, AMD 29K 55
file name, logical 36, 38
files, including 40
files, input 5
`fill` directive 38
filling memory 45
`float` directive 39
`float` directive, i386 90
`float` directive, M680x0 81
`float` directive, VAX 50
floating point numbers 18
floating point numbers (double) 37
floating point numbers (single) 39, 44
floating point, AMD 29K (ieee) 55
floating point, H8/300 (ieee) 57
floating point, H8/500 (ieee) 63
floating point, HPPA (ieee) 67
floating point, i386 90
floating point, i960 (ieee) 76
floating point, M680x0 81
floating point, SH (ieee) 72
floating point, SPARC (ieee) 84
floating point, VAX 50
flonums 18

format of error messages 7
format of warning messages 6
formfeed (\f) 16
functions, in expressions 32
fwait instruction, i386 90

G

gbr960, i960 postprocessor 75
gfloat directive, VAX 50
global 94
global directive 39
gp register, MIPS 100
grouping data 24

H

H8/300 addressing modes 56
H8/300 floating point (ieee) 57
H8/300 line comment character 56
H8/300 line separator 56
H8/300 machine directives (none) 58
H8/300 opcode summary 58
H8/300 options (none) 56
H8/300 registers 56
H8/300 size suffixes 61
H8/300 support 56
H8/300H, assembling for 58
H8/500 addressing modes 63
H8/500 floating point (ieee) 63
H8/500 line comment character 62
H8/500 line separator 62
H8/500 machine directives (none) 63
H8/500 opcode summary 63
H8/500 options (none) 62
H8/500 registers 62
H8/500 support 62
half directive, SPARC 84
hex character code (\xdd) 16
hexadecimal integers 18
hfloat directive, VAX 50
HPPA directives not supported 67
HPPA floating point (ieee) 67
HPPA Syntax 66
HPPA-only directives 67
hword directive 39

I

i386 16-bit code 91
i386 conversion instructions 87
i386 floating point 90
i386 fwait instruction 90
i386 immediate operands 86
i386 jump optimization 89
i386 jump, call, return 86
i386 jump/call operands 86
i386 memory references 88
i386 mul, imul instructions 91
i386 opcode naming 86
i386 opcode prefixes 88
i386 options (none) 86
i386 register operands 86
i386 registers 87
i386 sections 86
i386 size suffixes 86
i386 source, destination operands 86
i386 support 86
i386 syntax compatibility 86
i80306 support 86
i960 architecture options 75
i960 branch recording 75
i960 callj pseudo-opcode 77
i960 compare and jump expansions . . . 78
i960 compare/branch instructions 78
i960 floating point (ieee) 76
i960 machine directives 76
i960 opcodes 77
i960 options 75
i960 support 75
ident directive 39
identifiers, AMD 29K 54
if directive 40
ifdef directive 40
ifndef directive 40
ifnotdef directive 40
immediate character, M680x0 83
immediate character, VAX 53
immediate operands, i386 86
imul instruction, i386 91
include directive 40
include directive search path 10
indirect character, VAX 53
infix operators 32
inhibiting interrupts, i386 88
input 5
input file linenumbers 6
instruction set, M680x0 82

instruction summary, H8/300 58
instruction summary, H8/500 63
instruction summary, SH 72
instruction summary, Z8000 95
instructions and directives 15
int directive 40
int directive, H8/300 58
int directive, H8/500 63
int directive, i386 90
int directive, SH 72
integer expressions 31
integer, 16-byte 42
integer, 8-byte 43
integers 17
integers, 16-bit 39
integers, 32-bit 40
integers, binary 17
integers, decimal 17
integers, hexadecimal 18
integers, octal 17
integers, one byte 36
internal as sections 24
invocation summary 1

J

joining text and data sections 11
jump instructions, i386 87
jump optimization, i386 89
jump/call operands, i386 86

L

label (:) 15
labels 27
lcomm directive 40
ld 6
leafproc directive, i960 76
length of symbols 15
lflags directive (ignored) 41
line comment character 14
line comment character, AMD 29K 54
line comment character, H8/300 56
line comment character, H8/500 62
line comment character, M680x0 83
line comment character, SH 71
line comment character, Z8000 93
line directive 41
line directive, AMD 29K 55
line numbers, in input files 6
line numbers, in warnings/errors 7
line separator character 15
line separator, AMD 29K 54
line separator, H8/300 56
line separator, H8/500 62
line separator, SH 71
line separator, Z8000 93
lines starting with # 14
linker 6
linker, and assembler 21
list directive 41
listing control, turning off 42
listing control, turning on 41
listing control: new page 37
listing control: paper size 43
listing control: subtitle 43
listing control: title line 46
listings, enabling 9
little endian output, MIPS 3
little-endian output, MIPS 99
ln directive 41
local common symbols 40
local labels, retaining in output 10
local symbol names 28
location counter 28
location counter, advancing 42
logical file name 36, 38
logical line number 41
logical line numbers 14
long directive 42
long directive, i386 90
lval 94

M

M680x0 addressing modes 79, 80
M680x0 architecture options 79
M680x0 branch improvement 82
M680x0 directives 81
M680x0 floating point 81
M680x0 immediate character 83
M680x0 line comment character 83
M680x0 opcodes 82
M680x0 options 79
M680x0 pseudo-opcodes 82
M680x0 size modifiers 79
M680x0 support 79
M680x0 syntax 79, 80

machine dependencies 49
machine directives, AMD 29K 55
machine directives, H8/300 (none) 58
machine directives, H8/500 (none) 63
machine directives, i960 76
machine directives, SH (none) 72
machine directives, SPARC 84
machine directives, VAX 50
machine independent directives 35
machine instructions (not covered) 4
machine-independent syntax 13
manual, structure and purpose 4
memory references, i386 88
merging text and data sections 11
messages from `as` 6
minus, permitted arguments 32
MIPS architecture options 99
MIPS big-endian output 99
MIPS debugging directives 100
MIPS ECOFF sections 100
MIPS endianness 3
MIPS ISA 3
MIPS ISA override 101
MIPS little-endian output 99
MIPS R2000 99
MIPS R3000 99
MIPS R4000 99
MIPS R6000 99
mit 79
mnemonics for opcodes, VAX 51
mnemonics, H8/300 58
mnemonics, H8/500 63
mnemonics, SH 72
mnemonics, Z8000 95
Motorola syntax for the 680x0 80
`mul` instruction, i386 91
multi-line statements 15

N

`name` 94
named section (COFF) 44
named sections 23
names, symbol 27
naming object file 11
new page, in listings 37
newline (`\n`) 16
newline, required at file end 15
`nolist` directive 42
null-terminated strings 36
number constants 17
numbered subsections 24
numbers, 16-bit 39
numeric values 31

O

object file 6
object file format 5
object file name 11
object file, after errors 12
obsolescent directives 48
`octa` directive 42
octal character code (`\`*`ddd`*) 16
octal integers 17
opcode mnemonics, VAX 51
opcode naming, i386 86
opcode prefixes, i386 88
opcode suffixes, i386 86
opcode summary, H8/300 58
opcode summary, H8/500 63
opcode summary, SH 72
opcode summary, Z8000 95
opcodes for AMD 29K 55
opcodes, i960 77
opcodes, M680x0 82
operand delimiters, i386 86
operand notation, VAX 53
operands in expressions 31
operator precedence 32
operators, in expressions 32
operators, permitted arguments 32
option summary 1
options for AMD29K (none) 54
options for i386 (none) 86
options for SPARC 84
options for VAX/VMS 50
options, all versions of `as` 9
options, command line 5
options, H8/300 (none) 56
options, H8/500 (none) 62
options, i960 75
options, M680x0 79
options, SH (none) 71
options, Z8000 93
`org` directive 42
other attribute, of `a.out` symbol 30
output file 6

P

padding the location counter 35
page, in listings 37
paper size, for listings 43
paths for .include 10
patterns, writing in memory 38
plus, permitted arguments 32
precedence of operators 32
precision, floating point 18
prefix operators 32
prefixes, i386 88
preprocessing 13
preprocessing, turning on and off 13
primary attributes, COFF symbols 30
proc directive, SPARC 84
protected registers, AMD 29K 54
pseudo-opcodes, M680x0 82
pseudo-ops for branch, VAX 51
pseudo-ops, machine independent 35
psize directive 43
purpose of gnu as 4

Q

quad directive 43
quad directive, i386 90

R

real-mode code, i386 91
register names, AMD 29K 54
register names, H8/300 56
register names, VAX 53
register operands, i386 86
registers, H8/500 62
registers, i386 87
registers, SH 71
registers, Z8000 93
relocation 21
relocation example 23
repeat prefixes, i386 88
reserve directive, SPARC 84
return instructions, i386 86
rsect 95

S

sbttl directive 43
scl directive 43
search path for .include 10
sect directive, AMD 29K 55
section directive 44
section override prefixes, i386 88
section-relative addressing 22
sections 21
sections in messages, internal 24
sections, i386 86
sections, named 23
seg directive, SPARC 84
segm 94
set directive 44
SH addressing modes 71
SH floating point (ieee) 72
SH line comment character 71
SH line separator 71
SH machine directives (none) 72
SH opcode summary 72
SH options (none) 71
SH registers 71
SH support 71
short directive 44
single character constant 17
single directive 44
single directive, i386 90
sixteen bit integers 39
sixteen byte integer 42
size directive 44
size modifiers, M680x0 79
size prefixes, i386 88
size suffixes, H8/300 61
sizes operands, i386 86
skip directive, M680x0 82
skip directive, SPARC 85
small objects, MIPS ECOFF 100
SOM symbol attributes 30
source program 5
source, destination operands; i386 86
space directive 45
space used, maximum for assembly ... 11
SPARC architectures 84
SPARC floating point (ieee) 84
SPARC machine directives 84
SPARC options 84
SPARC support 84
special characters, M680x0 83
special purpose registers, AMD 29K .. 54
stabd directive 45
stabn directive 46

stabs directive 46
stabx directives 45
standard as sections 21
standard input, as input file 5
statement on multiple lines 15
statement separator character 15
statement separator, AMD 29K 54
statement separator, H8/300 56
statement separator, H8/500 62
statement separator, SH 71
statement separator, Z8000 93
statements, structure of 15
statistics, about assembly 11
stopping the assembly 35
string constants 16
string directive 46
string directive on HPPA 69
string literals 36
string, copying to object file 46
structure debugging, COFF 46
subexpressions 31
subtitles for listings 43
subtraction, permitted arguments 32
summary of options 1
support 66
supporting files, including 40
suppressing warnings 11
sval 94
symbol attributes 29
symbol attributes, a.out 29
symbol attributes, COFF 30
symbol attributes, SOM 30
symbol descriptor, COFF 37
symbol names 27
symbol names, '$' in 62, 71
symbol names, local 28
symbol names, temporary 28
symbol storage class (COFF) 43
symbol type 29
symbol type, COFF 47
symbol value 29
symbol value, setting 44
symbol values, assigning 27
symbol, common 36
symbol, making visible to linker 39
symbolic debuggers, information for 45
symbols 27
symbols with lowercase, VAX/VMS 50
symbols, assigning values to 38
symbols, local common 40
syntax compatibility, i386 86
syntax, M680x0 .... 79, 80
syntax, machine-independent 13
sysproc directive, i960 77

T

tab (\t) 16
tag directive 46
temporary symbol names 28
text and data sections, joining 11
text directive 46
text section 23
tfloat directive, i386 90
time, total for assembly 11
title directive 46
trusted compiler 10
turning preprocessing on and off 13
type directive 47
type of a symbol 29

U

undefined section 23
unsegm 94
use directive, AMD 29K 55

V

val directive 47
value attribute, COFF 47
value of a symbol 29
VAX bitfields not supported 53
VAX branch improvement 51
VAX command-line options ignored 49
VAX displacement sizing character 53
VAX floating point 50
VAX immediate character 53
VAX indirect character 53
VAX machine directives 50
VAX opcode mnemonics 51
VAX operand notation 53
VAX register names 53
VAX support 49
Vax-11 C compatibility 50
VAX/VMS options 50
version of as 11
VMS (VAX) options 50

W

warning for altered difference tables . . 10
warning messages 6
warnings, suppressing 11
whitespace 13
whitespace, removed by preprocessor 13
wide floating point directives, VAX 50
`word` directive 47
`word` directive, H8/300 58
`word` directive, H8/500 63
`word` directive, i386 90
`word` directive, SH 72
`word` directive, SPARC 85
writing patterns in memory 38
`wval` 94

X

`xword` directive, SPARC 85

Z

Z800 addressing modes 94
Z8000 directives 94
Z8000 line comment character 93
Z8000 line separator 93
Z8000 opcode summary 95
Z8000 options 93
Z8000 registers 93
Z8000 support 93
zero-terminated strings 36

GASP, an assembly preprocessor

for GASP version 1

March 1994

Roland Pesch

Cygnus Support

Copyright © 1994 Free Software Foundation, Inc.

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

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

Table of Contents

1 What is GASP? 1

2 Command Line Options 3

3 Preprocessor Commands 5
3.1 Conditional assembly 5
3.2 Repetitive sections of assembly 6
3.3 Preprocessor variables 7
3.4 Defining your own directives 9
3.5 Data output 11
3.5.1 Initialized data 11
3.5.2 Uninitialized data 12
3.6 Assembly listing control 13
3.7 Miscellaneous commands 13
3.8 Details of the GASP syntax 14
3.8.1 Special syntactic markers 15
3.8.2 String and numeric constants 16
3.8.3 Symbols 16
3.8.4 Arithmetic expressions in GASP 16
3.8.5 String primitives 17
3.9 Alternate macro syntax 17

Index 19

1 What is GASP?

The primary purpose of the GNU assembler is to assemble the output of other programs—notably compilers. When you have to hand-code specialized routines in assembly, that means the GNU assembler is an unfriendly processor: it has no directives for macros, conditionals, or many other conveniences that you might expect.

In some cases you can simply use the C preprocessor, or a generalized preprocessor like M4; but this can be awkward, since none of these things are designed with assembly in mind.

GASP fills this need. It is expressly designed to provide the facilities you need with hand-coded assembly code. Implementing it as a preprocessor, rather than part of the assembler, allows the maximum flexibility: you can use it with hand-coded assembly, without paying a penalty of added complexity in the assembler you use for compiler output.

Here is a small example to give the flavor of GASP. This input to GASP

```
        .MACRO  saveregs from=8 to=14
count   .ASSIGNA \from
        ! save r\from..r\to
        .AWHILE  \&count LE \to
        mov     r\&count,@-sp
count   .ASSIGNA  \&count + 1
        .AENDW
        .ENDM

        saveregs from=12

bar:    mov     #H'dead+10,r0
foo     .SDATAC "hello"<10>
        .END
```

generates this assembly program:

```
        ! save r12..r14
        mov     r12,@-sp
        mov     r13,@-sp
        mov     r14,@-sp

bar:    mov     #57005+10,r0
foo:    .byte   6,104,101,108,108,111,10
```

2 Command Line Options

The simplest way to use GASP is to run it as a filter and assemble its output. In Unix and its ilk, you can do this, for example:

```
$ gasp prog.asm | as -o prog.o
```

Naturally, there are also a few command-line options to allow you to request variations on this basic theme. Here is the full set of possibilities for the GASP command line.

```
gasp  [ -a | --alternate ]
      [ -c char | --commentchar char ]
      [ -d | --debug ]  [ -h | --help ]
      [ -o outfile | --output outfile ]
      [ -p | --print ]  [ -s | --copysource ]
      [ -u | --unreasonable ]  [ -v | --version ]
      infile ...
```

`infile` ...
: The input file names. You must specify at least one input file; if you specify more, GASP preprocesses them all, concatenating the output in the order you list the *infile* arguments.

  Mark the end of each input file with the preprocessor command `.END`. See Section 3.7 "Miscellaneous commands," page 13.

`-a`
`--alternate`
: Use alternative macro syntax. See Section 3.9 "Alternate macro syntax," page 17, for a discussion of how this syntax differs from the default GASP syntax.

`-c 'char'`
`--commentchar 'char'`
: Use *char* as the comment character. The default comment character is '`!`'. For example, to use a semicolon as the comment character, specify '`-c ';'`' on the GASP command line. Since assembler command characters often have special significance to command shells, it is a good idea to quote or escape *char* when you specify a comment character.

  For the sake of simplicity, all examples in this manual use the default comment character '`!`'.

`-d`
`--debug`
: Show debugging statistics. In this version of GASP, this option produces statistics about the string buffers that GASP allocates internally. For each defined buffersize *s*, GASP shows the number of strings *n* that it allocated, with a line like this:

`strings size s : n`

GASP displays these statistics on the standard error stream, when done preprocessing.

`-h`
`--help` Display a summary of the GASP command line options.

`-o outfile`
`--output outfile`
Write the output in a file called *outfile*. If you do not use the '-o' option, GASP writes its output on the standard output stream.

`-p`
`--print` Print line numbers. GASP obeys this option *only* if you also specify '-s' to copy source lines to its output. With '-s -p', GASP displays the line number of each source line copied (immediately after the comment character at the beginning of the line).

`-s`
`--copysource`
Copy the source lines to the output file. Use this option to see the effect of each preprocessor line on the GASP output. GASP places a comment character ('!' by default) at the beginning of each source line it copies, so that you can use this option and still assemble the result.

`-u`
`--unreasonable`
Bypass "unreasonable expansion" limit. Since you can define GASP macros inside other macro definitions, the preprocessor normally includes a sanity check. If your program requires more than 1,000 nested expansions, GASP normally exits with an error message. Use this option to turn off this check, allowing unlimited nested expansions.

`-v`
`--version`
Display the GASP version number.

3 Preprocessor Commands

GASP commands have a straightforward syntax that fits in well with assembly conventions. In general, a command extends for a line, and may have up to three fields: an optional label, the command itself, and optional arguments to the command. You can write commands in upper or lower case, though this manual shows them in upper case. See Section 3.8 "Details of the GASP syntax," page 14, for more information.

3.1 Conditional assembly

The conditional-assembly directives allow you to include or exclude portions of an assembly depending on how a pair of expressions, or a pair of strings, compare.

The overall structure of conditionals is familiar from many other contexts. `.AIF` marks the start of a conditional, and precedes assembly for the case when the condition is true. An optional `.AELSE` precedes assembly for the converse case, and an `.AENDI` marks the end of the condition.

You may nest conditionals up to a depth of 100; GASP rejects nesting beyond that, because it may indicate a bug in your macro structure.

Conditionals are primarily useful inside macro definitions, where you often need different effects depending on argument values. See Section 3.4 "Defining your own directives," page 9, for details about defining macros.

`.AIF` *expra cmp exprb*
`.AIF "`*stra*`"` *cmp* `"`*strb*`"`

The governing condition goes on the same line as the `.AIF` preprocessor command. You may compare either two strings, or two expressions.

When you compare strings, only two conditional *cmp* comparison operators are available: '`EQ`' (true if *stra* and *strb* are identical), and '`NE`' (the opposite).

When you compare two expressions, *both expressions must be absolute* (see Section 3.8.4 "Arithmetic expressions in GASP," page 16). You can use these *cmp* comparison operators with expressions:

`EQ` Are *expra* and *exprb* equal? (For strings, are *stra* and *strb* identical?)

`NE` Are *expra* and *exprb* different? (For strings, are *stra* and *strb* different?

`LT` Is *expra* less than *exprb*? (Not allowed for strings.)

`LE` Is *expra* less than or equal to *exprb*? (Not allowed for strings.)

`GT` Is *expra* greater than *exprb*? (Not allowed for strings.)

`GE` Is *expra* greater than or equal to *exprb*? (Not allowed for strings.)

`.AELSE` Marks the start of assembly code to be included if the condition fails. Optional, and only allowed within a conditional (between `.AIF` and `.AENDI`).

`.AENDI` Marks the end of a conditional assembly.

3.2 Repetitive sections of assembly

Two preprocessor directives allow you to repeatedly issue copies of the same block of assembly code.

`.AREPEAT` *aexp*
`.AENDR` If you simply need to repeat the same block of assembly over and over a fixed number of times, sandwich one instance of the repeated block between `.AREPEAT` and `.AENDR`. Specify the number of copies as *aexp* (which must be an absolute expression). For example, this repeats two assembly statements three times in succession:

```
        .AREPEAT        3
        rotcl   r2
        div1    r0,r1
        .AENDR
```

`.AWHILE` *expra cmp exprb*
`.AENDW`
`.AWHILE` *stra cmp strb*
`.AENDW` To repeat a block of assembly depending on a conditional test, rather than repeating it for a specific number of times, use `.AWHILE`. `.AENDW` marks the end of the repeated block. The conditional comparison works exactly the same way as for `.AIF`, with the same comparison operators (see Section 3.1 “Conditional assembly,” page 5).

Since the terms of the comparison must be absolute expression, `.AWHILE` is primarily useful within macros. See Section 3.4 “Defining your own directives,” page 9.

You can use the `.EXITM` preprocessor directive to break out of loops early (as well as to break out of macros). See Section 3.4 “Defining your own directives,” page 9.

3.3 Preprocessor variables

You can use variables in GASP to represent strings, registers, or the results of expressions.

You must distinguish two kinds of variables:

1. Variables defined with `.EQU` or `.ASSIGN`. To evaluate this kind of variable in your assembly output, simply mention its name. For example, these two lines define and use a variable ‘`eg`’:

```
eg      .EQU    FLIP-64
        . . .
        mov.l   eg,r0
```

   Do not use this kind of variable in conditional expressions or while loops; GASP only evaluates these variables when writing assembly output.

2. Variables for use during preprocessing. You can define these with `.ASSIGNC` or `.ASSIGNA`. To evaluate this kind of variable, write ‘`\&`’ before the variable name; for example,

```
opcit   .ASSIGNA   47
        . . .
        .AWHILE   \&opcit GT 0
        . . .
        .AENDW
```

   GASP treats macro arguments almost the same way, but to evaluate them you use the prefix ‘`\`’ rather than ‘`\&`’. See Section 3.4 “Defining your own directives,” page 9.

pvar `.EQU` *expr*

Assign preprocessor variable *pvar* the value of the expression *expr*. There are no restrictions on redefinition; use ‘`.EQU`’ with the same *pvar* as often as you find it convenient.

pvar .ASSIGN *expr*
Almost the same as .EQU, save that you may not redefine *pvar* using .ASSIGN once it has a value.

pvar .ASSIGNA *aexpr*
Define a variable with a numeric value, for use during preprocessing. *aexpr* must be an absolute expression. You can redefine variables with .ASSIGNA at any time.

pvar .ASSIGNC "*str*"
Define a variable with a string value, for use during preprocessing. You can redefine variables with .ASSIGNC at any time.

pvar .REG (*register*)
Use .REG to define a variable that represents a register. In particular, *register* is *not evaluated* as an expression. You may use .REG at will to redefine register variables.

All these directives accept the variable name in the "label" position, that is at the left margin. You may specify a colon after the variable name if you wish; the first example above could have started 'eg:' with the same effect.

3.4 Defining your own directives

The commands `.MACRO` and `.ENDM` allow you to define macros that generate assembly output. You can use these macros with a syntax similar to built-in GASP or assembler directives. For example, this definition specifies a macro `SUM` that adds together a range of consecutive registers:

```
        .MACRO  SUM FROM=0, TO=9
        ! \FROM \TO
        mov     r\FROM,r10
COUNT   .ASSIGNA        \FROM+1
        .AWHILE \&COUNT LE \TO
        add     r\&COUNT,r10
COUNT   .ASSIGNA        \&COUNT+1
        .AENDW
        .ENDM
```

With that definition, '`SUM 0,5`' generates this assembly output:

```
        ! 0 5
        mov     r0,r10
        add     r1,r10
        add     r2,r10
        add     r3,r10
        add     r4,r10
        add     r5,r10
```

`.MACRO` *`macname`*
`.MACRO` *`macname macargs`* `. . .`

Begin the definition of a macro called *`macname`*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with '`=`*`deflt`*'. For example, these are all valid `.MACRO` statements:

`.MACRO COMM`

Begin the definition of a macro called `COMM`, which takes no arguments.

`.MACRO PLUS1 P, P1`
`.MACRO PLUS1 P P1`

Either statement begins the definition of a macro called `PLUS1`, which takes two arguments; within

the macro definition, write '\P' or '\P1' to evaluate the arguments.

.MACRO RESERVE_STR P1=0 P2
: Begin the definition of a macro called RESERVE_STR, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as 'RESERVE_STR *a*,*b*' (with '\P1' evaluating to *a* and '\P2' evaluating to *b*), or as 'RESERVE_STR ,*b*' (with '\P1' evaluating as the default, in this case '0', and '\P2' evaluating to *b*).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, 'SUM 9,17' is equivalent to 'SUM TO=17, FROM=9'. Macro arguments are preprocessor variables similar to the variables you define with '.ASSIGNA' or '.ASSIGNC'; in particular, you can use them in conditionals or for loop control. (The only difference is the prefix you write to evaluate the variable: for a macro argument, write '*argname*', but for a preprocessor variable, write '\&*varname*'.)

name .MACRO
name .MACRO (*macargs* . . .)
: An alternative form of introducing a macro definition: specify the macro name in the label position, and the arguments (if any) between parentheses after the name. Defaulting rules and usage work the same way as for the other macro definition syntax.

.ENDM
: Mark the end of a macro definition.

.EXITM
: Exit early from the current macro definition, .AREPEAT loop, or .AWHILE loop.

\@
: GASP maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with '\@', but *only within a macro definition*.

LOCAL *name* [, . . .]
: *Warning:* LOCAL *is only available if you select "alternate macro syntax" with '*-a*' or '*--alternate*'.* See Section 3.9 "Alternate macro syntax," page 17.

  Generate a string replacement for each of the *name* arguments, and replace any instances of *name* in each macro expansion. The replacement string is unique in the assembly,

and different for each separate macro expansion. `LOCAL` allows you to write macros that define symbols, without fear of conflict between separate macro expansions.

3.5 Data output

In assembly code, you often need to specify working areas of memory; depending on the application, you may want to initialize such memory or not. GASP provides preprocessor directives to help you avoid repetitive coding for both purposes.

You can use labels as usual to mark the data areas.

3.5.1 Initialized data

These are the GASP directives for initialized data, and the standard GNU assembler directives they expand to:

`.DATA` *expr*, *expr*, . . .
`.DATA.B` *expr*, *expr*, . . .
`.DATA.W` *expr*, *expr*, . . .
`.DATA.L` *expr*, *expr*, . . .

Evaluate arithmetic expressions *expr*, and emit the corresponding `as` directive (labelled with *lab*). The unqualified `.DATA` emits '`.long`'; `.DATA.B` emits '`.byte`'; `.DATA.W` emits '`.short`'; and `.DATA.L` emits '`.long`'.

For example, '`foo .DATA 1,2,3`' emits '`foo: .long 1,2,3`'.

`.DATAB` *repeat*, *expr*
`.DATAB.B` *repeat*, *expr*
`.DATAB.W` *repeat*, *expr*
`.DATAB.L` *repeat*, *expr*

Make `as` emit *repeat* copies of the value of the expression *expr* (using the `as` directive `.fill`). '`.DATAB.B`' repeats one-byte values; '`.DATAB.W`' repeats two-byte values; and '`.DATAB.L`' repeats four-byte values. '`.DATAB`' without a suffix repeats four-byte values, just like '`.DATAB.L`'.

repeat must be an absolute expression with a positive value.

`.SDATA "`*str*`" . . .`

String data. Emits a concatenation of bytes, precisely as you specify them (in particular, *nothing is added to mark the end* of the string). See Section 3.8.2 "String and numeric constants," page 16, for details about how to write strings. `.SDATA` concatenates multiple arguments, making it easy to

switch between string representations. You can use commas to separate the individual arguments for clarity, if you choose.

`.SDATAB` *repeat*`, "`*str*`" ...`
: Repeated string data. The first argument specifies how many copies of the string to emit; the remaining arguments specify the string, in the same way as the arguments to `.SDATA`.

`.SDATAZ "`*str*`" ...`
: Zero-terminated string data. Just like `.SDATA`, except that `.SDATAZ` writes a zero byte at the end of the string.

`.SDATAC "`*str*`" ...`
: Count-prefixed string data. Just like `.SDATA`, except that GASP precedes the string with a leading one-byte count. For example, '`.SDATAC "HI"`' generates '`.byte 2,72,73`'. Since the count field is only one byte, you can only use `.SDATAC` for strings less than 256 bytes in length.

3.5.2 Uninitialized data

Use the `.RES`, `.SRES`, `.SRESC`, and `.SRESZ` directives to reserve memory and leave it uninitialized. GASP resolves these directives to appropriate calls of the GNU `as .space` directive.

`.RES` *count*
`.RES.B` *count*
`.RES.W` *count*
`.RES.L` *count*
: Reserve room for *count* uninitialized elements of data. The suffix specifies the size of each element: `.RES.B` reserves *count* bytes, `.RES.W` reserves *count* pairs of bytes, and `.RES.L` reserves *count* quartets. `.RES` without a suffix is equivalent to `.RES.L`.

`.SRES` *count*
`.SRES.B` *count*
`.SRES.W` *count*
`.SRES.L` *count*
: `.SRES` is a synonym for '`.RES`'.

`.SRESC` *count*
`.SRESC.B` *count*
`.SRESC.W` *count*
`.SRESC.L` *count*
: Like `.SRES`, but reserves space for *count*`+1` elements.

`.SRESZ` *count*
`.SRESZ.B` *count*
`.SRESZ.W` *count*
`.SRESZ.L` *count*

Like `.SRES`, but reserves space for *count*+1 elements.

3.6 Assembly listing control

The GASP listing-control directives correspond to related GNU `as` directives.

`.PRINT LIST`
`.PRINT NOLIST`

Print control. This directive emits the GNU `as` directive `.list` or `.nolist`, according to its argument. See section "`.list`" in *Using as*, for details on how these directives interact.

`.FORM LIN=`*ln*
`.FORM COL=`*cols*
`.FORM LIN=`*ln* `COL=`*cols*

Specify the page size for assembly listings: *ln* represents the number of lines, and *cols* the number of columns. You may specify either page dimension independently, or both together. If you do not specify the number of lines, GASP assumes 60 lines; if you do not specify the number of columns, GASP assumes 132 columns. (Any values you may have specified in previous instances of `.FORM` do *not* carry over as defaults.) Emits the `.psize` assembler directive.

`.HEADING` *string*

Specify *string* as the title of your assembly listings. Emits '`.title "`*string*`"`'.

`.PAGE` Force a new page in assembly listings. Emits '`.eject`'.

3.7 Miscellaneous commands

`.ALTERNATE`

Use the alternate macro syntax henceforth in the assembly. See Section 3.9 "Alternate macro syntax," page 17.

`.ORG` This command is recognized, but not yet implemented. GASP generates an error message for programs that use `.ORG`.

`.RADIX` *s* GASP understands numbers in any of base two, eight, ten, or sixteen. You can encode the base explicitly in any numeric

constant (see Section 3.8.2 “String and numeric constants,” page 16). If you write numbers without an explicit indication of the base, the most recent ‘`.RADIX s`’ command determines how they are interpreted. *s* is a single letter, one of the following:

`.RADIX B` Base 2.

`.RADIX Q` Base 8.

`.RADIX D` Base 10. This is the original default radix.

`.RADIX H` Base 16.

You may specify the argument *s* in lower case (any of ‘`bqdh`’) with the same effects.

`.EXPORT` *name*
`.GLOBAL` *name*
Declare *name* global (emits ‘`.global` *name*’). The two directives are synonymous.

`.PROGRAM` No effect: GASP accepts this directive, and silently ignores it.

`.END` Mark end of each preprocessor file. GASP issues a warning if it reaches end of file without seeing this command.

`.INCLUDE "`*str*`"`
Preprocess the file named by *str*, as if its contents appeared where the `.INCLUDE` directive does. GASP imposes a maximum limit of 30 stacked include files, as a sanity check.

`.ALIGN` *size*
Evaluate the absolute expression *size*, and emit the assembly instruction ‘`.align` *size*’ using the result.

3.8 Details of the GASP syntax

Since GASP is meant to work with assembly code, its statement syntax has no surprises for the assembly programmer.

Whitespace (blanks or tabs; *not* newline) is partially significant, in that it delimits up to three fields in a line. The amount of whitespace does not matter; you may line up fields in separate lines if you wish, but GASP does not require that.

The *first field*, an optional *label*, must be flush left in a line (with no leading whitespace) if it appears at all. You may use a colon after the label if you wish; GASP neither requires the colon nor objects to it (but will not include it as part of the label name).

The *second field*, which must appear after some whitespace, contains a GASP or assembly *directive*.

Any *further fields* on a line are *arguments* to the directive; you can separate them from one another using either commas or whitespace.

3.8.1 Special syntactic markers

GASP recognizes a few special markers: to delimit comments, to continue a statement on the next line, to separate symbols from other characters, and to copy text to the output literally. (One other special marker, '\@', works only within macro definitions; see Section 3.4 "Defining your own directives," page 9.)

The trailing part of any GASP source line may be a *comment*. A comment begins with the first unquoted comment character ('!' by default), or an escaped or doubled comment character ('\!' or '!!' by default), and extends to the end of a line. You can specify what comment character to use with the '-c' option (see Chapter 2 "Command Line Options," page 3). The two kinds of comment markers lead to slightly different treatment:

`!`
: A single, un-escaped comment character generates an assembly comment in the GASP output. GASP evaluates any preprocessor variables (macro arguments, or variables defined with .ASSIGNA or .ASSIGNC) present. For example, a macro that begins like this

```
.MACRO  SUM FROM=0, TO=9
! \FROM \TO
```

issues as the first line of output a comment that records the values you used to call the macro.

`\!`
`!!`
: Either an escaped comment character, or a double comment character, marks a GASP source comment. GASP does not copy such comments to the assembly output.

To *continue a statement* on the next line of the file, begin the second line with the character '+'.

Occasionally you may want to prevent GASP from preprocessing some particular bit of text. To *copy literally* from the GASP source to its output, place '\(' before the string to copy, and ')' at the end. For example, write '\(\!)' if you need the characters '\!' in your assembly output.

To *separate a preprocessor variable* from text to appear immediately after its value, write a single quote ('). For example, '.SDATA "\P'1"' writes a string built by concatenating the value of P and the digit '1'. (You cannot achieve this by writing just '\P1', since 'P1' is itself a valid name for a preprocessor variable.)

3.8.2 String and numeric constants

There are two ways of writing *string constants* in GASP: as literal text, and by numeric byte value. Specify a string literal between double quotes (`"str"`). Specify an individual numeric byte value as an absolute expression between angle brackets (`<expr>`. Directives that output strings allow you to specify any number of either kind of value, in whatever order is convenient, and concatenate the result. (Alternate syntax mode introduces a number of alternative string notations; see Section 3.9 "Alternate macro syntax," page 17.)

You can write *numeric constants* either in a specific base, or in whatever base is currently selected (either 10, or selected by the most recent `.RADIX`).

To write a number in a *specific base*, use the pattern `s'ddd`: a base specifier character `s`, followed by a single quote followed by digits `ddd`. The base specifier character matches those you can specify with `.RADIX`: '`B`' for base 2, '`Q`' for base 8, '`D`' for base 10, and '`H`' for base 16. (You can write this character in lower case if you prefer.)

3.8.3 Symbols

GASP recognizes symbol names that start with any alphabetic character, '_', or '`$`', and continue with any of the same characters or with digits. Label names follow the same rules.

3.8.4 Arithmetic expressions in GASP

There are two kinds of expressions, depending on their result: *absolute* expressions, which resolve to a constant (that is, they do not involve any values unknown to GASP), and *relocatable* expressions, which must reduce to the form

```
addsym+const-subsym
```

where `addsym` and `subsym` are assembly symbols of unknown value, and `const` is a constant.

Arithmetic for GASP expressions follows very similar rules to C. You can use parentheses to change precedence; otherwise, arithmetic primitives have decreasing precedence in the order of the following list.

1. Single-argument `+` (identity), `-` (arithmetic opposite), or `~` (bitwise negation). *The argument must be an absolute expression.*
2. `*` (multiplication) and `/` (division). *Both arguments must be absolute expressions.*
3. `+` (addition) and `-` (subtraction). *At least one argument must be absolute.*

4. `&` (bitwise and). *Both arguments must be absolute.*
5. `|` (bitwise or) and `~` (bitwise exclusive or; `^` in C). *Both arguments must be absolute.*

3.8.5 String primitives

You can use these primitives to manipulate strings (in the argument field of GASP statements):

`.LEN("str")`
: Calculate the length of string `"str"`, as an absolute expression. For example, '`.RES.B .LEN("sample")`' reserves six bytes of memory.

`.INSTR("string", "seg", ix)`
: Search for the first occurrence of *seg* after position *ix* of *string*. For example, '`.INSTR("ABCDEFG", "CDE", 0)`' evaluates to the absolute result `2`.

  The result is `-1` if *seg* does not occur in *string* after position *ix*.

`.SUBSTR("string",start,len)`
: The substring of *string* beginning at byte number *start* and extending for *len* bytes.

3.9 Alternate macro syntax

If you specify '`-a`' or '`--alternate`' on the GASP command line, the preprocessor uses somewhat different syntax. This syntax is reminiscent of the syntax of Phar Lap macro assembler, but it is *not* meant to be a full emulation of Phar Lap or similar assemblers. In particular, GASP does not support directives such as `DB` and `IRP`, even in alternate syntax mode.

In particular, '`-a`' (or '`--alternate`') elicits these differences:

Preprocessor directives
: You can use GASP preprocessor directives without a leading '.' dot. For example, you can write '`SDATA`' with the same effect as '`.SDATA`'.

LOCAL
: One additional directive, `LOCAL`, is available. See Section 3.4 "Defining your own directives," page 9, for an explanation of how to use `LOCAL`.

String delimiters

You can write strings delimited in these other ways besides `"string"`:

`'string'` You can delimit strings with single-quote charaters.

`<string>` You can delimit strings with matching angle brackets.

single-character string escape

To include any single character literally in a string (even if the character would otherwise have some special meaning), you can prefix the character with '`!`' (an exclamation mark). For example, you can write '`<4.3 !> 5.4!!>`' to get the literal text '`4.3 > 5.4!`'.

Expression results as strings

You can write '`%expr`' to evaluate the expression `expr` and use the result as a string.

Index

!

! default comment char 3

-

--alternate 3
--commentchar *'char'* 3
--copysource 4
--debug 3
--help 4
--output *outfile* 4
--print 4
--unreasonable 4
--version 4
-a 3
-c *'char'* 3
-d 3
-h 4
-o *outfile* 4
-p 4
-s 4
-u 4
-v 4

.

.AELSE 6
.AENDI 6
.AENDR 6
.AENDW 6
.AIF "*stra*" *cmp* "*strb*" 5
.AIF *expra cmp exprb* 5
.ALIGN *size* 14
.ALTERNATE 13
.AREPEAT *aexp* 6
.AWHILE *expra cmp exprb* 6
.AWHILE *stra cmp strb* 6
.DATA *expr*, *expr*, ... 11
.DATA.B *expr*, *expr*, ... 11
.DATA.L *expr*, *expr*, ... 11
.DATA.W *expr*, *expr*, ... 11
.DATAB *repeat*, *expr* 11
.DATAB.B *repeat*, *expr* 11
.DATAB.L *repeat*, *expr* 11
.DATAB.W *repeat*, *expr* 11
.END 14
.ENDM 10
.EXITM 10
.EXPORT *name* 14
.FORM COL=*cols* 13
.FORM LIN=*ln* 13
.FORM LIN=*ln* COL=*cols* 13
.GLOBAL *name* 14
.HEADING *string* 13
.INCLUDE "*str*" 14
.INCLUDE
.INSTR("*string*", "*seg*", *ix*) 17
.LEN("*str*") 17
.MACRO *macname* 9
.MACRO *macname macargs* 9
.ORG 13
.PAGE 13
.PRINT LIST 13
.PRINT NOLIST 13
.PROGRAM 14
.RADIX *s* 13
.RES *count* 12
.RES.B *count* 12
.RES.L *count* 12
.RES.W *count* 12
.SDATA "*str*" 11
.SDATAB *repeat*, "*str*" 12
.SDATAC "*str*" 12
.SDATAZ "*str*" 12
.SRES *count* 12
.SRES.B *count* 12
.SRES.L *count* 12
.SRES.W *count* 12
.SRESC *count* 12
.SRESC.B *count* 12
.SRESC.L *count* 12
.SRESC.W *count* 12
.SRESZ *count* 13
.SRESZ.B *count* 13
.SRESZ.L *count* 13
.SRESZ.W *count* 13
.SUBSTR("*string*",*start*,*len*) ... 17

;

`;` as comment char 3

+

`+` 15

\

`\@` 10

A

absolute expressions 16
argument fields 15
avoiding preprocessing 15

B

bang, as comment 3
breaking out of loops 7

C

comment character, changing 3
comments 15
continuation character 15
copying literally to output 15

D

directive field 14

E

`EQ` 5
exclamation mark, as comment 3

F

fields of gasp source line 14

G

`GE` 6
`GT` 6

I

infile 3

L

label field 14
`LE` 6
literal copy to output 15
`LOCAL` *name* `[ , ... ]` 10
loops, breaking out of 7
`LT` 6

M

macros, count executed 10

N

name `.MACRO` 10
name `.MACRO (` *macargs* `... )` .. 10
`NE` 5
number of macros executed 10

P

preprocessing, avoiding 15
pvar `.ASSIGN` *expr* 8
pvar `.ASSIGNA` *aexpr* 8
pvar `.ASSIGNC "`*str*`"` 8
pvar `.EQU` *expr* 7
pvar `.REG (`*register*`)` 8

R

relocatable expressions 16

S

semicolon, as comment 3
shriek, as comment 3
symbol separator 15
symbols, separating from text 15

T

text, separating from symbols 15

W

whitespace 14

Using ld

The GNU linker

`ld` version 2
January 1994

Steve Chamberlain and Roland Pesch
Cygnus Support

Cygnus Support
steve@cygnus.com, pesch@cygnus.com
Using LD, the GNU linker
Edited by Jeffrey Osier (jeffrey@cygnus.com)
and Roland Pesch (pesch@cygnus.com)

Copyright © 1991, 1992, 1993, 1994 1995 Free Software Foundation, Inc.

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

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

Table of Contents

1 Overview 1

2 Invocation 3
- 2.1 Command Line Options 3
- 2.2 Environment Variables 13

3 Command Language 15
- 3.1 Linker Scripts 15
- 3.2 Expressions 15
  - 3.2.1 Integers 16
  - 3.2.2 Symbol Names 16
  - 3.2.3 The Location Counter 17
  - 3.2.4 Operators 18
  - 3.2.5 Evaluation 18
  - 3.2.6 Assignment: Defining Symbols 18
  - 3.2.7 Arithmetic Functions 20
- 3.3 Memory Layout 22
- 3.4 Specifying Output Sections 23
  - 3.4.1 Section Definitions 24
  - 3.4.2 Section Placement 25
  - 3.4.3 Section Data Expressions 27
  - 3.4.4 Optional Section Attributes 29
- 3.5 The Entry Point 31
- 3.6 Option Commands 32

4 Machine Dependent Features 35
- 4.1 `ld` and the H8/300 35
- 4.2 `ld` and the Intel 960 family 35

5 BFD 37
- 5.1 How it works: an outline of BFD 37
  - 5.1.1 Information Loss 38
  - 5.1.2 The BFD canonical object-file format 38

Appendix A MRI Compatible Script Files 41

Index 45

1 Overview

`ld` combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run `ld`.

`ld` accepts Linker Command Language files written in a superset of AT&T's Link Editor Command Language syntax, to provide explicit and total control over the linking process.

This version of `ld` uses the general purpose BFD libraries to operate on object files. This allows `ld` to read, combine, and write object files in many different formats—for example, COFF or `a.out`. Different formats may be linked together to produce any available kind of object file. See Chapter 5 "BFD," page 37, for more information.

Aside from its flexibility, the GNU linker is more helpful than other linkers in providing diagnostic information. Many linkers abandon execution immediately upon encountering an error; whenever possible, `ld` continues executing, allowing you to identify other errors (or, in some cases, to get an output file in spite of the error).

2 Invocation

The GNU linker `ld` is meant to cover a broad range of situations, and to be as compatible as possible with other linkers. As a result, you have many choices to control its behavior.

2.1 Command Line Options

Here is a summary of the options you can use on the `ld` command line:

```
ld [ -o output ]  objfile...
 [ -Aarchitecture ] [ -b input-format ] [ -Bstatic ]
 [ -c MRI-commandfile ] [ -d | -dc | -dp ]
 [ -defsym symbol=expression ]
 [ -dynamic-linker file ] [ -embedded-relocs ]
 [ -e entry ] [ -F ] [ -F format ]
 [ -format input-format ] [ -g ] [ -G size ] [ -help ]
 [ -i ] [ -larchive ] [ -Lsearchdir ] [ -M ]
 [ -Map mapfile ] [ -m emulation ] [ -N | -n ]
 [ -noinhibit-exec ] [ -no-keep-memory ] [ -oformat output-format ]
 [ -R filename ] [ -relax ] [ -retain-symbols-file filename ]
 [ -r | -Ur ] [ -rpath dir ] [ -S ] [ -s ] [ -soname name ]
 [ -sort-common ] [ -stats ] [ -T commandfile ]
 [ -Ttext org ] [ -Tdata org ]
 [ -Tbss org ] [ -t ] [ -traditional-format ]
 [ -u symbol] [-V] [-v] [ -verbose] [ -version ]
 [ -warn-common ] [ -warn-once ] [ -y symbol ] [ -X ]  [-x ]
 [ -( [ archives ] -) ] [ --start-group [ archives ] --end-group ]
 [ -split-by-reloc count ] [ -split-by-file ] [ --whole-archive ]
```

This plethora of command-line options may seem intimidating, but in actual practice few of them are used in any particular context.
For instance, a frequent use of `ld` is to link standard Unix object files on a standard, supported Unix system. On such a system, to link a file `hello.o`:

```
ld -o output /lib/crt0.o hello.o -lc
```

This tells `ld` to produce a file called *`output`* as the result of linking the file `/lib/crt0.o` with `hello.o` and the library `libc.a`, which will come from the standard search directories. (See the discussion of the '`-l`' option below.)

The command-line options to `ld` may be specified in any order, and may be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option.

The exceptions—which may meaningfully be used more than once—are '`-A`', '`-b`' (or its synonym '`-format`'), '`-defsym`', '`-L`', '`-l`', '`-R`', '`-u`', and '`-(`' (or its synonym '`--start-group`')..

The list of object files to be linked together, shown as *objfile*..., may follow, precede, or be mixed in with command-line options, except that an *objfile* argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but you can specify other forms of binary input files using '-l', '-R', and the script command language. If *no* binary input files at all are specified, the linker does not produce any output, and issues the message 'No input files'.

If the linker can not recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link (either the default linker script or the one specified by using '-T'). This feature permits the linker to link against a file which appears to be an object or an archive, but actually merely defines some symbol values, or uses INPUT or GROUP to load other objects. See Chapter 3 "Commands," page 15.

For options whose names are a single letter, option arguments must either follow the option letter without intervening whitespace, or be given as separate arguments immediately following the option that requires them.

For options whose names are multiple letters, either one dash or two can precede the option name; for example, '--oformat' and '-oformat' are equivalent. Arguments to multiple-letter options must either be separated from the option name by an equals sign, or be given as separate arguments immediately following the option that requires them. For example, '--oformat srec' and '--oformat=srec' are equivalent. Unique abbreviations of the names of multiple-letter options are accepted.

-A*architecture*

In the current release of ld, this option is useful only for the Intel 960 family of architectures. In that ld configuration, the *architecture* argument identifies the particular architecture in the 960 family, enabling some safeguards and modifying the archive-library search path. See Section 4.2 "ld and the Intel 960 family," page 35, for details.

Future releases of ld may support similar functionality for other architecture families.

-b *input-format*

ld may be configured to support more than one kind of object file. If your ld is configured this way, you can use the '-b' option to specify the binary format for input object files that follow this option on the command line. Even when ld is configured to support alternative object formats, you don't usually need to specify this, as ld should be config-

ured to expect as a default input format the most usual format on each machine. *input-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) '`-format` *input-format*' has the same effect, as does the script command `TARGET`. See Chapter 5 "BFD," page 37.

You may want to use this option if you are linking files with an unusual binary format. You can also use '`-b`' to switch formats explicitly (when linking object files of different formats), by including '`-b` *input-format*' before each group of object files in a particular format.

The default format is taken from the environment variable `GNUTARGET`. See Section 2.2 "Environment," page 13. You can also define the input format from a script, using the command `TARGET`; see Section 3.6 "Option Commands," page 32.

`-Bstatic` Do not link against shared libraries. This option is accepted for command-line compatibility with the SunOS linker.

`-c` *MRI-commandfile*
: For compatibility with linkers produced by MRI, `ld` accepts script files written in an alternate, restricted command language, described in Appendix A "MRI Compatible Script Files," page 41. Introduce MRI script files with the option '`-c`'; use the '`-T`' option to run linker scripts written in the general-purpose `ld` scripting language. If *MRI-cmdfile* does not exist, `ld` looks for it in the directories specified by any '`-L`' options.

`-d`
`-dc`
`-dp` These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with '`-r`'). The script command `FORCE_COMMON_ALLOCATION` has the same effect. See Section 3.6 "Option Commands," page 32.

`-defsym` *symbol=expression*
: Create a global symbol in the output file, containing the absolute address given by *expression*. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *expression* in this context: you may give a hexadecimal constant or the name of an existing symbol, or use `+` and `-` to add or subtract hexadecimal constants or symbols. If you

need more elaborate expressions, consider using the linker command language from a script (see Section 3.2.6 "Assignment: Symbol Definitions," page 18). *Note:* there should be no white space between `symbol`, the equals sign ("`=`"), and `expression`.

`-dynamic-linker file`
: Set the name of the dynamic linker. This is only meaningful when generating dynamically linked ELF executables. The default dynamic linker is normally correct; don't use this unless you know what you are doing.

`-embedded-relocs`
: This option is only meaningful when linking MIPS embedded PIC code, generated by the -membedded-pic option to the GNU compiler and assembler. It causes the linker to create a table which may be used at runtime to relocate any data which was statically initialized to pointer values. See the code in testsuite/ld-empic for details.

`-e entry`
: Use `entry` as the explicit symbol for beginning execution of your program, rather than the default entry point. See Section 3.5 "Entry Point," page 31, for a discussion of defaults and other ways of specifying the entry point.

`-F`
`-Fformat`
: Ignored. Some older linkers used this option throughout a compilation toolchain for specifying object-file format for both input and output object files. The mechanisms `ld` uses for this purpose (the '`-b`' or '`-format`' options for input files, '`-oformat`' option or the `TARGET` command in linker scripts for output files, the `GNUTARGET` environment variable) are more flexible, but `ld` accepts the '`-F`' option for compatibility with scripts written to call the old linker.

`-format input-format`
: Synonym for '`-b input-format`'.

`-g`
: Ignored. Provided for compatibility with other tools.

`-Gvalue`
`-G value`
: Set the maximum size of objects to be optimized using the GP register to `size` under MIPS ECOFF. Ignored for other object file formats.

`-help`
: Print a summary of the command-line options on the standard output and exit.

`-i`
: Perform an incremental link (same as option '`-r`').

`-l`*`ar`*
: Add archive file *`archive`* to the list of files to link. This option may be used any number of times. `ld` will search its path-list for occurrences of `lib`*`ar`*`.a` for every *`archive`* specified.

`-L`*`searchdir`*
`-L` *`searchdir`*
: Add path *`searchdir`* to the list of paths that `ld` will search for archive libraries and `ld` control scripts. You may use this option any number of times.

  The default set of paths searched (without being specified with '`-L`') depends on which emulation mode `ld` is using, and in some cases also on how it was configured. See Section 2.2 "Environment," page 13.

  The paths can also be specified in a link script with the `SEARCH_DIR` command.

`-M`
: Print (to the standard output) a link map—diagnostic information about where symbols are mapped by `ld`, and information on global common storage allocation.

`-Map` *`mapfile`*
: Print to the file *`mapfile`* a link map—diagnostic information about where symbols are mapped by `ld`, and information on global common storage allocation.

`-m`*`emulation`*
`-m` *`emulation`*
: Emulate the *`emulation`* linker. You can list the available emulations with the '`--verbose`' or '`-V`' options. The default depends on how your `ld` was configured.

`-N`
: Set the text and data sections to be readable and writable. Also, do not page-align the data segment. If the output format supports Unix style magic numbers, mark the output as `OMAGIC`.

`-n`
: Set the text segment to be read only, and mark the output as `NMAGIC` if possible.

`-noinhibit-exec`
: Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever.

`-no-keep-memory`
: `ld` normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This op-

tion tells `ld` to instead optimize for memory usage, by rereading the symbol tables as necessary. This may be required if `ld` runs out of memory space while linking a large executable.

`-o` *output*
: Use *output* as the name for the program produced by `ld`; if this option is not specified, the name '`a.out`' is used by default. The script command `OUTPUT` can also specify the output file name.

`-oformat` *output-format*
: `ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`-oformat`' option to specify the binary format for the output object file. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be configured to produce as a default output format the most usual format on each machine. *output-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it. See Chapter 5 "BFD," page 37.

`-R` *filename*
: Read symbol names and their addresses from *filename*, but do not relocate it or include it in the output. This allows your output file to refer symbolically to absolute locations of memory defined in other programs.

`-relax`
: An option with machine dependent effects. Currently this option is only supported on the H8/300 and the Intel 960. See Section 4.1 "`ld` and the H8/300," page 35. See Section 4.2 "`ld` and the Intel 960 family," page 35.

  On some platforms, the '`-relax`' option performs global optimizations that become possible when the linker resolves addressing in the program, such as relaxing address modes and synthesizing new instructions in the output object file.

  On platforms where this is not supported, '`-relax`' is accepted, but ignored.

`-retain-symbols-file` *filename*
: Retain *only* the symbols listed in the file *filename*, discarding all others. *filename* is simply a flat file, with one symbol name per line. This option is especially useful in environments (such as VxWorks) where a large global symbol table is accumulated gradually, to conserve run-time memory.

'-retain-symbols-file' does *not* discard undefined symbols, or symbols needed for relocations.

You may only specify '-retain-symbols-file' once in the command line. It overrides '-s' and '-S'.

-rpath *dir*
: Add a directory to the runtime library search path. This is only meaningful when linking an ELF executable with shared objects. All -rpath arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime.

-r
: Generate relocatable output—i.e., generate an output file that can in turn serve as input to ld. This is often called *partial linking*. As a side effect, in environments that support standard Unix magic numbers, this option also sets the output file's magic number to OMAGIC. If this option is not specified, an absolute file is produced. When linking C++ programs, this option *will not* resolve references to constructors; to do that, use '-Ur'.

This option does the same thing as '-i'.

-S
: Omit debugger symbol information (but not all symbols) from the output file.

-s
: Omit all symbol information from the output file.

-soname *name*
: When creating an ELF shared object, set the internal DT_SONAME field to the specified name. When an executable is linked with a shared object which has a DT_SONAME field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the DT_SONAME field rather than the using the file name given to the linker.

-sort-common
: Normally, when ld places the global common symbols in the appropriate output sections, it sorts them by size. First come all the one byte symbols, then all the two bytes, then all the four bytes, and then everything else. This is to prevent gaps between symbols due to alignment constraints. This option disables that sorting.

-split-by-reloc *count*
: Trys to creates extra sections in the output file so that no single output section in the file contains more than *count* relocations. This is useful when generating huge relocatable

for downloading into certain real time kernels with the COFF object file format; since COFF cannot represent more than 65535 relocations in a single section. Note that this will fail to work with object file formats which do not support arbitrary sections. The linker will not split up individual input sections for redistribution, so if a single input section contains more than *count* relocations one output section will contain that many relocations.

`-split-by-file`
: Similar to -split-by-reloc but creates a new output section for each input file.

`-stats`
: Compute and display statistics about the operation of the linker, such as execution time and memory usage.

`-Tbss` *org*
`-Tdata` *org*
`-Ttext` *org*
: Use *org* as the starting address for—respectively—the `bss`, `data`, or the `text` segment of the output file. *org* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '`0x`' usually associated with hexadecimal values.

`-T` *commandfile*
`-T`*commandfile*
: Read link commands from the file *commandfile*. These commands replace `ld`'s default link script (rather than adding to it), so *commandfile* must specify everything necessary to describe the target format. See Chapter 3 "Commands," page 15. If *commandfile* does not exist, `ld` looks for it in the directories specified by any preceding '`-L`' options. Multiple '`-T`' options accumulate.

`-t`
: Print the names of the input files as `ld` processes them.

`-traditional-format`
: For some targets, the output of `ld` is different in some ways from the output of some existing linker. This switch requests `ld` to use the traditional format instead.

For example, on SunOS, `ld` combines duplicate entries in the symbol string table. This can reduce the size of an output file with full debugging information by over 30 percent. Unfortunately, the SunOS `dbx` program can not read the resulting program (`gdb` has no trouble). The '`-traditional-format`' switch tells `ld` to not combine duplicate entries.

`-u symbol`
: Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. '`-u`' may be repeated with different option arguments to enter additional undefined symbols.

`-Ur`
: For anything other than C++ programs, this option is equivalent to '`-r`': it generates relocatable output—i.e., an output file that can in turn serve as input to `ld`. When linking C++ programs, '`-Ur`' *does* resolve references to constructors, unlike '`-r`'. It does not work to use '`-Ur`' on files that were themselves linked with '`-Ur`'; once the constructor table has been built, it cannot be added to. Use '`-Ur`' only for the last partial link, and '`-r`' for the others.

`--verbose`
: Display the version number for `ld` and list the linker emulations supported. Display which input files can and cannot be opened.

`-v`
`-V`
: Display the version number for `ld`. The `-V` option also lists the supported emulations.

`-version`
: Display the version number for `ld` and exit.

`-warn-common`
: Warn when a common symbol is combined with another common symbol or with a symbol definition. Unix linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get some warnings about symbols in the libraries as well as in your programs.

  There are three kinds of global symbols, illustrated here by C examples:

  '`int i = 1;`'
  : A definition, which goes in the initialized data section of the output file.

  '`extern int i;`'
  : An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.

'int i;' A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file. The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.

The '-warn-common' option can produce five kinds of warnings. Each warning consists of a pair of lines: the first describes the symbol just encountered, and the second describes the previous symbol encountered with the same name. One or both of the two symbols will be a common symbol.

1. Turning a common symbol into a reference, because there is already a definition for the symbol.

```
file(section): warning: common of 'symbol'
   overridden by definition
file(section): warning: defined here
```

2. Turning a common symbol into a reference, because a later definition for the symbol is encountered. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: definition of 'symbol'
   overriding common
file(section): warning: common is here
```

3. Merging a common symbol with a previous same-sized common symbol.

```
file(section): warning: multiple common
   of 'symbol'
file(section): warning: previous common is here
```

4. Merging a common symbol with a previous larger common symbol.

```
file(section): warning: common of 'symbol'
   overridden by larger common
file(section): warning: larger common is here
```

5. Merging a common symbol with a previous smaller common symbol. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: common of 'symbol'
   overriding smaller common
file(section): warning: smaller common is here
```

`-warn-once`
: Only warn once for each undefined symbol, rather than once per module which refers to it.

For each archive mentioned on the command line, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library.

`-X`
: Delete all temporary local symbols. For most targets, this is all local symbols whose names begin with '`L`'.

`-x`
: Delete all local symbols.

`-y` *`symbol`*
: Print the name of each linked file in which *`symbol`* appears. This option may be given any number of times. On many systems it is necessary to prepend an underscore.

This option is useful when you have an undefined symbol in your link but don't know where the reference is coming from.

`-(` *`archives`* `-)`
`--start-group` *`archives`* `--end-group`
: The *`archives`* should be a list of archive files. They may be either explicit file names, or '`-l`' options.

The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved.

Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives.

2.2 Environment Variables

You can change the behavior of `ld` with the environment variable `GNUTARGET`.

`GNUTARGET` determines the input-file object format if you don't use '`-b`' (or its synonym '`-format`'). Its value should be one of the BFD names for an input format (see Chapter 5 "BFD," page 37). If there is no `GNUTARGET` in

the environment, `ld` uses the natural format of the target. If `GNUTARGET` is set to `default` then BFD attempts to discover the input format by examining binary input files; this method often succeeds, but there are potential ambiguities, since there is no method of ensuring that the magic number used to specify object-file formats is unique. However, the configuration procedure for BFD on each system places the conventional format for that system first in the search-list, so ambiguities are resolved in favor of convention.

3 Command Language

The command language provides explicit control over the link process, allowing complete specification of the mapping between the linker's input files and its output. It controls:

- input files
- file formats
- output file layout
- addresses of sections
- placement of common blocks

You may supply a command file (also known as a link script) to the linker either explicitly through the '-T' option, or implicitly as an ordinary file. If the linker opens a file which it cannot recognize as a supported object or archive format, it reports an error.

3.1 Linker Scripts

The ld command language is a collection of statements; some are simple keywords setting a particular option, some are used to select and group input files or name output files; and two statement types have a fundamental and pervasive impact on the linking process.

The most fundamental command of the ld command language is the SECTIONS command (see Section 3.4 "SECTIONS," page 23). Every meaningful command script must have a SECTIONS command: it specifies a "picture" of the output file's layout, in varying degrees of detail. No other command is required in all cases.

The MEMORY command complements SECTIONS by describing the available memory in the target architecture. This command is optional; if you don't use a MEMORY command, ld assumes sufficient memory is available in a contiguous block for all output. See Section 3.3 "MEMORY," page 22.

You may include comments in linker scripts just as in C: delimited by '/*' and '*/'. As in C, comments are syntactically equivalent to whitespace.

3.2 Expressions

Many useful commands involve arithmetic expressions. The syntax for expressions in the command language is identical to that of C expressions, with the following features:

- All expressions evaluated as integers and are of "long" or "unsigned long" type.

- All constants are integers.
- All of the C arithmetic operators are provided.
- You may reference, define, and create global variables.
- You may call special purpose built-in functions.

3.2.1 Integers

An octal integer is '0' followed by zero or more of the octal digits ('01234567').

```
_as_octal = 0157255;
```

A decimal integer starts with a non-zero digit followed by zero or more digits ('0123456789').

```
_as_decimal = 57005;
```

A hexadecimal integer is '0x' or '0X' followed by one or more hexadecimal digits chosen from '0123456789abcdefABCDEF'.

```
_as_hex = 0xdead;
```

To write a negative integer, use the prefix operator '-'; see Section 3.2.4 "Operators," page 18.

```
_as_neg = -57005;
```

Additionally the suffixes K and M may be used to scale a constant by 1024 or 1024^2 respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

3.2.2 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or point and may include any letters, underscores, digits, points, and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword, by surrounding the symbol name in double quotes:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, 'A-B' is one symbol, whereas 'A - B' is an expression involving subtraction.

3.2.3 The Location Counter

The special linker variable *dot* '`.`' always contains the current output location counter. Since the `.` always refers to a location in an output section, it must always appear in an expression within a `SECTIONS` command. The `.` symbol may appear anywhere that an ordinary symbol is allowed in an expression, but its assignments have a side effect. Assigning a value to the `.` symbol will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
{
  output :
  {
  file1(.text)
  . = . + 1000;
  file2(.text)
  . += 1000;
  file3(.text)
  } = 0x1234;
}
```

In the previous example, `file1` is located at the beginning of the output section, then there is a 1000 byte gap. Then `file2` appears, also with a 1000 byte gap following before `file3` is loaded. The notation '`= 0x1234`' specifies what data to write in the gaps (see Section 3.4.4 "Section Options," page 29).

3.2.4 Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels:

<table>
<tr><th>Precedence</th><th>Associativity</th><th>Operators</th></tr>
<tr><td>highest</td><td></td><td></td></tr>
<tr><td>1</td><td>left</td><td>- ~ ! †</td></tr>
<tr><td>2</td><td>left</td><td>* / %</td></tr>
<tr><td>3</td><td>left</td><td>+ -</td></tr>
<tr><td>4</td><td>left</td><td>>> <<</td></tr>
<tr><td>5</td><td>left</td><td>== != > < <= >=</td></tr>
<tr><td>6</td><td>left</td><td>&</td></tr>
<tr><td>7</td><td>left</td><td>|</td></tr>
<tr><td>8</td><td>left</td><td>&&</td></tr>
<tr><td>9</td><td>left</td><td>||</td></tr>
<tr><td>10</td><td>right</td><td>? :</td></tr>
<tr><td>11</td><td>right</td><td>&= += -= *= /= ‡</td></tr>
<tr><td>lowest</td><td></td><td></td></tr>
</table>

† Prefix operators.
‡ See Section 3.2.6 “Assignment,” page 18.

3.2.5 Evaluation

The linker uses “lazy evaluation” for expressions; it only calculates an expression when absolutely necessary. The linker needs the value of the start address, and the lengths of memory regions, in order to do any linking at all; these values are computed as soon as possible when the linker reads in the command file. However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

3.2.6 Assignment: Defining Symbols

You may create global symbols, and assign values (addresses) to global symbols, using any of the C assignment operators:

symbol = *expression* ;
symbol &= *expression* ;
symbol += *expression* ;
symbol -= *expression* ;
symbol *= *expression* ;
symbol /= *expression* ;

Two things distinguish assignment from other operators in `ld` expressions.

- Assignment may only be used at the root of an expression; '`a=b+3;`' is allowed, but '`a+b=3;`' is an error.
- You must place a trailing semicolon ("`;`") at the end of an assignment statement.

Assignment statements may appear:

- as commands in their own right in an `ld` script; or
- as independent statements within a `SECTIONS` command; or
- as part of the contents of a section definition in a `SECTIONS` command.

The first two cases are equivalent in effect—both define a symbol with an absolute address. The last case defines a symbol whose address is relative to a particular section (see Section 3.4 "SECTIONS," page 23).

When a linker expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file; a relocatable expression type is one in which the value is expressed as a fixed offset from the base of a section.

The type of the expression is controlled by its position in the script file. A symbol assigned within a section definition is created relative to the base of the section; a symbol assigned in any other place is created as an absolute symbol. Since a symbol created within a section definition is relative to the base of the section, it will remain relocatable if relocatable output is requested. A symbol may be created with an absolute value even when assigned to within a section definition by using the absolute assignment function `ABSOLUTE`. For example, to create an absolute symbol whose address is the last byte of an output section named `.data`:

```
SECTIONS{ ...
  .data :
    {
      *(.data)
      _edata = ABSOLUTE(.) ;
    }
... }
```

The linker tries to put off the evaluation of an assignment until all the terms in the source expression are known (see Section 3.2.5 "Evaluation,"

page 18). For instance, the sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation. Some expressions, such as those depending upon the location counter *dot*, '`.`' must be evaluated during allocation. If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following

```
SECTIONS { ...
  text 9+this_isnt_constant :
    { ...
    }
... }
```

will cause the error message "`Non constant expression for initial address`".

In some cases, it is desirable for a linker script to define a symbol only if it is referenced, and only if it is not defined by any object included in the link. For example, traditional linkers defined the symbol '`etext`'. However, ANSI C requires that the user be able to use '`etext`' as a function name without encountering an error. The `PROVIDE` keyword may be used to define a symbol, such as '`etext`', only if it is referenced but not defined. The syntax is `PROVIDE(`*`symbol`*` = `*`expression`*`)`.

3.2.7 Arithmetic Functions

The command language includes a number of built-in functions for use in link script expressions.

`ABSOLUTE(`*`exp`*`)`
: Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *`exp`*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section-relative.

`ADDR(`*`section`*`)`
: Return the absolute address of the named *`section`*. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
... }
```

`ALIGN(`*`exp`*`)`

Return the result of the current location counter (`.`) aligned to the next *exp* boundary. *exp* must be an expression whose value is a power of two. This is equivalent to

```
(. + exp - 1) & ~(exp - 1)
```

`ALIGN` doesn't change the value of the location counter—it just does arithmetic on it. As an example, to align the output `.data` section to the next `0x2000` byte boundary after the preceding section and to set a variable within the section to the next `0x8000` boundary after the input sections:

```
SECTIONS{ ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
... }
```

The first use of `ALIGN` in this example specifies the location of a section because it is used as the optional *start* attribute of a section definition (see Section 3.4.4 "Section Options," page 29). The second use simply defines the value of a variable.

The built-in `NEXT` is closely related to `ALIGN`.

`DEFINED(`*`symbol`*`)`

Return 1 if *symbol* is in the linker global symbol table and is defined, otherwise return 0. You can use this function to provide default values for symbols. For example, the following command-file fragment shows how to set a global symbol `begin` to the first location in the `.text` section—but if a symbol called `begin` already existed, its value is preserved:

```
SECTIONS{ ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
... }
```

`NEXT(exp)`
: Return the next unallocated address that is a multiple of *exp*. This function is closely related to `ALIGN(exp)`; unless you use the `MEMORY` command to define discontinuous memory for the output file, the two functions are equivalent.

`SIZEOF(section)`
: Return the size in bytes of the named *section*, if that section has been allocated. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
... }
```

`SIZEOF_HEADERS`
`sizeof_headers`
: Return the size in bytes of the output file's headers. You can use this number as the start address of the first section, if you choose, to facilitate paging.

3.3 Memory Layout

The linker's default configuration permits allocation of all available memory. You can override this configuration by using the `MEMORY` command. The `MEMORY` command describes the location and size of blocks of memory in the target. By using it carefully, you can describe which memory regions may be used by the linker, and which memory regions it must avoid. The linker does not shuffle sections to fit into the available regions, but does move the requested sections into the correct regions and issue errors when the regions become too full.

A command file may contain at most one use of the `MEMORY` command; however, you can define as many blocks of memory within it as you wish. The syntax is:

```
MEMORY
  {
    name (attr) : ORIGIN = origin, LENGTH = len
    ...
  }
```

name is a name used internally by the linker to refer to the region. Any symbol name may be used. The region names are stored in a separate name space, and will not conflict with symbols, file names or section names. Use distinct names to specify multiple regions.

(*attr*) is an optional list of attributes, permitted for compatibility with the AT&T linker but not used by `ld` beyond checking that the attribute list is valid. Valid attribute lists must be made up of the characters "`LIRWX`". If you omit the attribute list, you may omit the parentheses around it as well.

origin is the start address of the region in physical memory. It is an expression that must evaluate to a constant before memory allocation is performed. The keyword `ORIGIN` may be abbreviated to `org` or `o` (but not, for example, '`ORG`').

len is the size in bytes of the region (an expression). The keyword `LENGTH` may be abbreviated to `len` or `l`.

For example, to specify that memory has two regions available for allocation—one starting at 0 for 256 kilobytes, and the other starting at `0x40000000` for four megabytes:

```
MEMORY
  {
  rom : ORIGIN = 0, LENGTH = 256K
  ram : org = 0x40000000, l = 4M
  }
```

Once you have defined a region of memory named *mem*, you can direct specific output sections there by using a command ending in '`>`*mem*' within the `SECTIONS` command (see Section 3.4.4 "Section Options," page 29). If the combined output sections directed to a region are too big for the region, the linker will issue an error message.

3.4 Specifying Output Sections

The `SECTIONS` command controls exactly where input sections are placed into output sections, their order in the output file, and to which output sections they are allocated.

You may use at most one `SECTIONS` command in a script file, but you can have as many statements within it as you wish. Statements within the `SECTIONS` command can do one of three things:

- define the entry point;
- assign a value to a symbol;
- describe the placement of a named output section, and which input sections go into it.

You can also use the first two operations—defining the entry point and defining symbols—outside the `SECTIONS` command: see Section 3.5 “Entry Point,” page 31, and see Section 3.2.6 “Assignment,” page 18. They are permitted here as well for your convenience in reading the script, so that symbols and the entry point can be defined at meaningful points in your output-file layout.

If you do not use a `SECTIONS` command, the linker places each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file.

3.4.1 Section Definitions

The most frequently used statement in the `SECTIONS` command is the *section definition*, which specifies the properties of an output section: its location, alignment, contents, fill pattern, and target memory region. Most of these specifications are optional; the simplest form of a section definition is

```
SECTIONS { ...
  secname : {
    contents
  }
... }
```

secname is the name of the output section, and *contents* a specification of what goes there—for example, a list of input files or sections of input files (see Section 3.4.2 “Section Placement,” page 25). As you might assume, the whitespace shown is optional. You do need the colon ‘`:`’ and the braces ‘`{}`’, however.

secname must meet the constraints of your output format. In formats which only support a limited number of sections, such as `a.out`, the name must be one of the names supported by the format (`a.out`, for example, allows only `.text`, `.data` or `.bss`). If the output format supports any number of sections, but with numbers and not names (as is the case for Oasys), the name should be supplied as a quoted numeric string. A section name may consist of any sequence of characters, but any name

which does not conform to the standard `ld` symbol name syntax must be quoted. See Section 3.2.2 “Symbol Names,” page 16.

The linker will not create output sections which do not have any contents. This is for convenience when referring to input sections that may or may not exist. For example,

```
.foo { *(.foo }
```

will only create a ‘`.foo`’ section in the output file if there is a ‘`.foo`’ section in at least one input file.

3.4.2 Section Placement

In a section definition, you can specify the contents of an output section by listing particular input files, by listing particular input-file sections, or by a combination of the two. You can also place arbitrary data in the section, and define symbols relative to the beginning of the section.

The *contents* of a section definition may include any of the following kinds of statement. You can include as many of these as you like in a single section definition, separated from one another by whitespace.

filename You may simply name a particular input file to be placed in the current output section; *all* sections from that file are placed in the current section definition. If the file name has already been mentioned in another section definition, with an explicit section name list, then only those sections which have not yet been allocated are used.

To specify a list of particular files by name:

```
.data : { afile.o bfile.o cfile.o }
```

The example also illustrates that multiple statements can be included in the contents of a section definition, since each file name is a separate statement.

filename(*section*)
filename(*section*, *section*, . . .)
filename(*section section* . . .)

You can name one or more sections from your input files, for insertion in the current output section. If you wish to specify a list of input-file sections inside the parentheses, you may separate the section names by either commas or whitespace.

* (*section*)
* (*section*, *section*, . . .)
* (*section section* . . .)

Instead of explicitly naming particular input files in a link control script, you can refer to *all* files from the `ld` command

line: use '*' instead of a particular file name before the parenthesized input-file section list.

If you have already explicitly included some files by name, '*' refers to all *remaining* files—those whose places in the output file have not yet been defined.

For example, to copy sections 1 through 4 from an Oasys file into the .text section of an a.out file, and sections 13 and 14 into the .data section:

```
SECTIONS {
  .text :{
    *("1" "2" "3" "4")
  }

  .data :{
    *("13" "14")
  }
}
```

'[*section* . . .]' used to be accepted as an alternate way to specify named sections from all unallocated input files. Because some operating systems (VMS) allow brackets in file names, that notation is no longer supported.

filename(COMMON)
*(COMMON)

Specify where in your output file to place uninitialized data with this notation. *(COMMON) by itself refers to all uninitialized data from all input files (so far as it is not yet allocated); *filename*(COMMON) refers to uninitialized data from a particular file. Both are special cases of the general mechanisms for specifying where to place input-file sections: ld permits you to refer to uninitialized data as if it were in an input-file section named COMMON, regardless of the input file's format.

For example, the following command script arranges the output file into three consecutive sections, named .text, .data, and .bss, taking the input for each from the correspondingly named sections of all the input files:

```
SECTIONS {
  .text : { *(.text) }
  .data : { *(.data) }
  .bss :  { *(.bss)  *(COMMON) }
}
```

The following example reads all of the sections from file all.o and places them at the start of output section outputa which starts at location

0x10000. **All of section** .input1 **from file** foo.o **follows immediately, in the same output section. All of section** .input2 **from** foo.o **goes into output section** outputb**, followed by section** .input1 **from** foo1.o**. All of the remaining** .input1 **and** .input2 **sections from any files are written to output section** outputc.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

3.4.3 Section Data Expressions

The foregoing statements arrange, in your output file, data originating from your input files. You can also place data directly in an output section from the link command script. Most of these additional statements involve expressions; see Section 3.2 "Expressions," page 15. Although these statements are shown separately here for ease of presentation, no such segregation is needed within a section definition in the SECTIONS command; you can intermix them freely with any of the statements we've just described.

CREATE_OBJECT_SYMBOLS

Create a symbol for each input file in the current section, set to the address of the first byte of data written from that input file. For instance, with a.out files it is conventional to have a symbol for each input file. You can accomplish this by defining the output .text section as follows:

```
SECTIONS {
  .text 0x2020 :
     {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    _etext = ALIGN(0x2000);
    }
  ...
}
```

If `sample.ld` is a file containing this script, and `a.o`, `b.o`, `c.o`, and `d.o` are four input files with contents like the following—

```
/* a.c */

afunction() { }
int adata=1;
int abss;
```

'`ld -M -T sample.ld a.o b.o c.o d.o`' would create a map like this, containing symbols matching the object file names:

```
00000000 A __DYNAMIC
00004020 B _abss
00004000 D _adata
00002020 T _afunction
00004024 B _bbss
00004008 D _bdata
00002038 T _bfunction
00004028 B _cbss
00004010 D _cdata
00002050 T _cfunction
0000402c B _dbss
00004018 D _ddata
00002068 T _dfunction
00004020 D _edata
00004030 B _end
00004000 T _etext
00002020 t a.o
00002038 t b.o
00002050 t c.o
00002068 t d.o
```

`symbol = expression ;`
`symbol f= expression ;`

symbol is any symbol name (see Section 3.2.2 "Symbols," page 16). "*f*=" refers to any of the operators `&= += -= *= /=` which combine arithmetic and assignment.

When you assign a value to a symbol within a particular section definition, the value is relative to the beginning of

the section (see Section 3.2.6 “Assignment,” page 18). If you write

```
SECTIONS {
  abs = 14 ;
  . . .
  .data : { . . . rel = 14 ; . . . }
  abs2 = 14 + ADDR(.data);
  . . .
}
```

`abs` and `rel` do not have the same value; `rel` has the same value as `abs2`.

`BYTE(`*expression*`)`
`SHORT(`*expression*`)`
`LONG(`*expression*`)`
`QUAD(`*expression*`)`

By including one of these four statements in a section definition, you can explicitly place one, two, four, or eight bytes (respectively) at the current address of that section. `QUAD` is only supported when using a 64 bit host or target.

Multiple-byte quantities are represented in whatever byte order is appropriate for the output file format (see Chapter 5 “BFD,” page 37).

`FILL(`*expression*`)`

Specify the “fill pattern” for the current section. Any otherwise unspecified regions of memory within the section (for example, regions you skip over by assigning a new value to the location counter ‘`.`’) are filled with the two least significant bytes from the *expression* argument. A `FILL` statement covers memory locations *after* the point it occurs in the section definition; by including more than one `FILL` statement, you can have different fill patterns in different parts of an output section.

3.4.4 Optional Section Attributes

Here is the full syntax of a section definition, including all the optional portions:

```
SECTIONS {
. . .
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region =fill
. . .
}
```

secname and *contents* are required. See Section 3.4.1 "Section Definition," page 24, and see Section 3.4.2 "Section Placement," page 25 for details on *contents*. The remaining elements—*start*, BLOCK(*align*), (NOLOAD), AT (*ldadr*), >*region*, and =*fill*—are all optional.

start You can force the output section to be loaded at a specified address by specifying *start* immediately following the section name. *start* can be represented as any expression. The following example generates section *output* at location 0x40000000:

```
SECTIONS {
  ...
  output 0x40000000: {
    ...
    }
  ...
}
```

BLOCK(*align*)
: You can include BLOCK() specification to advance the location counter . prior to the beginning of the section, so that the section will begin at the specified alignment. *align* is an expression.

(NOLOAD) Use '(NOLOAD)' to prevent a section from being loaded into memory each time it is accessed. For example, in the script sample below, the ROM segment is addressed at memory location '0' and does not need to be loaded into each object file:

```
SECTIONS {
  ROM  0  (NOLOAD)  : { ... }
  ...
}
```

AT (*ldadr*)
: The expression *ldadr* that follows the AT keyword specifies the load address of the section. The default (if you do not use the AT keyword) is to make the load address the same as the relocation address. This feature is designed to make it easy to build a ROM image. For example, this SECTIONS definition creates two output sections: one called '.text', which starts at 0x1000, and one called '.mdata', which is loaded at the end of the '.text' section even though its relocation address is 0x2000. The symbol _data is defined with the value 0x2000:

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR(.text) + SIZEOF ( .text ) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

The run-time initialization code (for C programs, usually `crt0`) for use with a ROM generated this way has to include something like the following, to copy the initialized data from the ROM image to its runtime address:

```
char *src = _etext;
char *dst = _data;

/* ROM has data at end of text; copy it. */
while (dst < _edata) {
  *dst++ = *src++;
}

/* Zero bss */
for (dst = _bstart; dst< _bend; dst++)
  *dst = 0;
```

`>`*`region`*
: Assign this section to a previously defined region of memory. See Section 3.3 “MEMORY,” page 22.

`=`*`fill`*
: Including `=`*`fill`* in a section definition specifies the initial fill value for that section. You may use any expression to specify *`fill`*. Any unallocated holes in the current output section when written to the output file will be filled with the two least significant bytes of the value, repeated as necessary. You can also change the fill value with a `FILL` statement in the *`contents`* of a section definition.

3.5 The Entry Point

The linker command language includes a command specifically for defining the first executable instruction in an output file (its *entry point*). Its argument is a symbol name:

```
ENTRY(symbol)
```

Like symbol assignments, the `ENTRY` command may be placed either as an independent command in the command file, or among the section

definitions within the `SECTIONS` command—whatever makes the most sense for your layout.

`ENTRY` is only one of several ways of choosing the entry point. You may indicate it in any of the following ways (shown in descending order of priority: methods higher in the list override methods lower down).

- the '-e' *entry* command-line option;
- the `ENTRY(symbol)` command in a linker control script;
- the value of the symbol `start`, if present;
- the value of the symbol `_main`, if present;
- the address of the first byte of the `.text` section, if present;
- The address `0`.

For example, you can use these rules to generate an entry point with an assignment statement: if no symbol `start` is defined within your input files, you can simply define it, assigning it an appropriate value—

```
start = 0x2020;
```

The example shows an absolute address, but you can use any expression. For example, if your input object files use some other symbol-name convention for the entry point, you can just assign the value of whatever symbol contains the start address to `start`:

```
start = other_symbol ;
```

3.6 Option Commands

The command language includes a number of other commands that you can use for specialized purposes. They are similar in purpose to command-line options.

`CONSTRUCTORS`

This command ties up C++ style constructor and destructor records. The details of the constructor representation vary from one object format to another, but usually lists of constructors and destructors appear as special sections. The `CONSTRUCTORS` command specifies where the linker is to place the data from these sections, relative to the rest of the linked output. Constructor data is marked by the symbol `__CTOR_LIST__` at the start, and `__CTOR_LIST_END` at the end; destructor data is bracketed similarly, between `__DTOR_LIST__` and `__DTOR_LIST_END`. (The compiler must arrange to actually run this code; GNU C++ calls constructors from a subroutine `__main`, which it inserts automatically into the startup code for `main`, and destructors from `_exit`.)

`FLOAT`
`NOFLOAT`
: These keywords were used in some older linkers to request a particular math subroutine library. `ld` doesn't use the keywords, assuming instead that any necessary subroutines are in libraries specified using the general mechanisms for linking to archives; but to permit the use of scripts that were written for the older linkers, the keywords `FLOAT` and `NOFLOAT` are accepted and ignored.

`FORCE_COMMON_ALLOCATION`
: This command has the same effect as the '`-d`' command-line option: to make `ld` assign space to common symbols even if a relocatable output file is specified ('`-r`').

`INPUT (` *`file`*`,` *`file`*`, . . . )`
`INPUT (` *`file file`* `. . . )`
: Use this command to include binary input files in the link, without including them in a particular section definition. Specify the full name for each *`file`*, including '`.a`' if required.

  `ld` searches for each *`file`* through the archive-library search path, just as for files you specify on the command line. See the description of '`-L`' in Section 2.1 "Command Line Options," page 3.

  If you use '`-l`*`file`*', `ld` will transform the name to `lib`*`file`*`.a` as with the command line argument '`-l`'.

`GROUP (` *`file`*`,` *`file`*`, . . . )`
`GROUP (` *`file file`* `. . . )`
: This command is like `INPUT`, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of '`-(`' in Section 2.1 "Command Line Options," page 3.

`OUTPUT (` *`filename`* `)`
: Use this command to name the link output file *`filename`*. The effect of `OUTPUT(`*`filename`*`)` is identical to the effect of '`-o` *`filename`*', which overrides it. You can use this command to supply a default output-file name other than `a.out`.

`OUTPUT_ARCH (` *`bfdname`* `)`
: Specify a particular output machine architecture, with one of the names used by the BFD back-end routines (see Chapter 5 "BFD," page 37). This command is often unnecessary; the architecture is most often set implicitly by either the system BFD configuration or as a side effect of the `OUTPUT_FORMAT` command.

`OUTPUT_FORMAT ( bfdname )`
: When `ld` is configured to support multiple object code formats, you can use this command to specify a particular output format. *bfdname* is one of the names used by the BFD back-end routines (see Chapter 5 “BFD,” page 37). The effect is identical to the effect of the '`-oformat`' command-line option. This selection affects only the output file; the related command `TARGET` affects primarily input files.

`SEARCH_DIR ( path )`
: Add *path* to the list of paths where `ld` looks for archive libraries. `SEARCH_DIR(path)` has the same effect as '`-Lpath`' on the command line.

`STARTUP ( filename )`
: Ensure that *filename* is the first input file used in the link process.

`TARGET ( format )`
: When `ld` is configured to support multiple object code formats, you can use this command to change the input-file object code format (like the command-line option '`-b`' or its synonym '`-format`'). The argument *format* is one of the strings used by BFD to name binary formats. If `TARGET` is specified but `OUTPUT_FORMAT` is not, the last `TARGET` argument is also used as the default format for the `ld` output file. See Chapter 5 “BFD,” page 37.

  If you don't use the `TARGET` command, `ld` uses the value of the environment variable `GNUTARGET`, if available, to select the output file format. If that variable is also absent, `ld` uses the default format configured for your machine in the BFD libraries.

4 Machine Dependent Features

`ld` has additional features on some platforms; the following sections describe them. Machines where `ld` has no additional functionality are not listed.

4.1 `ld` and the H8/300

For the H8/300, `ld` can perform these global optimizations when you specify the '`-relax`' command-line option.

relaxing address modes
: `ld` finds all `jsr` and `jmp` instructions whose targets are within eight bits, and turns them into eight-bit program-counter relative `bsr` and `bra` instructions, respectively.

synthesizing instructions
: `ld` finds all `mov.b` instructions which use the sixteen-bit absolute address form, but refer to the top page of memory, and changes them to use the eight-bit address form. (That is: the linker turns '`mov.b @aa:16`' into '`mov.b @aa:8`' whenever the address *aa* is in the top page of memory).

4.2 `ld` and the Intel 960 family

You can use the '`-Aarchitecture`' command line option to specify one of the two-letter names identifying members of the 960 family; the option specifies the desired output target, and warns of any incompatible instructions in the input files. It also modifies the linker's search strategy for archive libraries, to support the use of libraries specific to each particular architecture, by including in the search loop names suffixed with the string identifying the architecture.

For example, if your `ld` command line included '`-ACA`' as well as '`-ltry`', the linker would look (in its built-in search paths, and in any paths you specify with '`-L`') for a library with the names

```
try
libtry.a
tryca
libtryca.a
```

The first two possibilities would be considered in any event; the last two are due to the use of '`-ACA`'.

You can meaningfully use '`-A`' more than once on a command line, since the 960 architecture family allows combination of target architectures;

each use will add another pair of name variants to search for when '-l' specifies a library.

ld supports the '-relax' option for the i960 family. If you specify '-relax', ld finds all balx and calx instructions whose targets are within 24 bits, and turns them into 24-bit program-counter relative bal and cal instructions, respectively. ld also turns cal instructions into bal instructions when it determines that the target subroutine is a leaf routine (that is, the target subroutine does not itself call any subroutines).

5 BFD

The linker accesses object and archive files using the BFD libraries. These libraries allow the linker to use the same routines to operate on object files whatever the object file format. A different object file format can be supported simply by creating a new BFD back end and adding it to the library. To conserve runtime memory, however, the linker and associated tools are usually configured to support only a subset of the object file formats available. You can use `objdump -i` (see section "objdump" in *The GNU Binary Utilities*) to list all the formats available for your configuration.

As with most implementations, BFD is a compromise between several conflicting requirements. The major factor influencing BFD design was efficiency: any time used converting between formats is time which would not have been spent had BFD not been involved. This is partly offset by abstraction payback; since BFD simplifies applications and back ends, more time and care may be spent optimizing algorithms for a greater speed.

One minor artifact of the BFD solution which you should bear in mind is the potential for information loss. There are two places where useful information can be lost using the BFD mechanism: during conversion and during output. See Section 5.1.1 "BFD information loss," page 38.

5.1 How it works: an outline of BFD

When an object file is opened, BFD subroutines automatically determine the format of the input object file. They then build a descriptor in memory with pointers to routines that will be used to access elements of the object file's data structures.

As different information from the the object files is required, BFD reads from different sections of the file and processes them. For example, a very common operation for the linker is processing symbol tables. Each BFD back end provides a routine for converting between the object file's representation of symbols and an internal canonical format. When the linker asks for the symbol table of an object file, it calls through a memory pointer to the routine from the relevant BFD back end which reads and converts the table into a canonical form. The linker then operates upon the canonical form. When the link is finished and the linker writes the output file's symbol table, another BFD back end routine is called to take the newly created symbol table and convert it into the chosen output format.

5.1.1 Information Loss

Information can be lost during output. The output formats supported by BFD do not provide identical facilities, and information which can be described in one form has nowhere to go in another format. One example of this is alignment information in `b.out`. There is nowhere in an `a.out` format file to store alignment information on the contained data, so when a file is linked from `b.out` and an `a.out` image is produced, alignment information will not propagate to the output file. (The linker will still use the alignment information internally, so the link is performed correctly).

Another example is COFF section names. COFF files may contain an unlimited number of sections, each one with a textual section name. If the target of the link is a format which does not have many sections (e.g., `a.out`) or has sections without names (e.g., the Oasys format), the link cannot be done simply. You can circumvent this problem by describing the desired input-to-output section mapping with the linker command language.

Information can be lost during canonicalization. The BFD internal canonical form of the external formats is not exhaustive; there are structures in input formats for which there is no direct representation internally. This means that the BFD back ends cannot maintain all possible data richness through the transformation between external to internal and back to external formats.

This limitation is only a problem when an application reads one format and writes another. Each BFD back end is responsible for maintaining as much data as possible, and the internal BFD canonical form has structures which are opaque to the BFD core, and exported only to the back ends. When a file is read in one format, the canonical form is generated for BFD and the application. At the same time, the back end saves away any information which may otherwise be lost. If the data is then written back in the same format, the back end routine will be able to use the canonical form provided by the BFD core as well as the information it prepared earlier. Since there is a great deal of commonality between back ends, there is no information lost when linking or copying big endian COFF to little endian COFF, or `a.out` to `b.out`. When a mixture of formats is linked, the information is only lost from the files whose format differs from the destination.

5.1.2 The BFD canonical object-file format

The greatest potential for loss of information occurs when there is the least overlap between the information provided by the source format, that stored by the canonical format, and that needed by the destination format. A brief description of the canonical form may help you

understand which kinds of data you can count on preserving across conversions.

files Information stored on a per-file basis includes target machine architecture, particular implementation format type, a demand pageable bit, and a write protected bit. Information like Unix magic numbers is not stored here—only the magic numbers' meaning, so a `ZMAGIC` file would have both the demand pageable bit and the write protected text bit set. The byte order of the target is stored on a per-file basis, so that big- and little-endian object files may be used with one another.

sections Each section in the input file contains the name of the section, the section's original address in the object file, size and alignment information, various flags, and pointers into other BFD data structures.

symbols Each symbol contains a pointer to the information for the object file which originally defined it, its name, its value, and various flag bits. When a BFD back end reads in a symbol table, it relocates all symbols to make them relative to the base of the section where they were defined. Doing this ensures that each symbol points to its containing section. Each symbol also has a varying amount of hidden private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. `ld` can operate on a collection of symbols of wildly different formats without problems.

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out`, type information is stored in the symbol table as long symbol names. This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away.

There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example, COFF, IEEE, Oasys) and the type is simple enough to fit within one word (nearly everything but aggregates), the information will be preserved.

relocation level
Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in, and a pointer to a relocation

type descriptor. Relocation is performed by passing messages through the relocation type descriptor and the symbol pointer. Therefore, relocations can be performed on output data using a relocation method that is only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation may be performed on a byte being written to a 68k COFF file, even though 68k COFF has no such relocation type.

line numbers

Object formats can contain, for debugging purposes, some form of mapping between symbols, source line numbers, and addresses in the output file. These addresses have to be relocated along with the symbol information. Each symbol with an associated list of line number records points to the first record of the list. The head of a line number list consists of a pointer to the symbol, which allows finding out the address of the function whose line number is being described. The rest of the list is made up of pairs: offsets into the section and line numbers. Any format which can simply derive this information can pass it successfully between formats (COFF, IEEE and Oasys).

Appendix A MRI Compatible Script Files

To aid users making the transition to GNU `ld` from the MRI linker, `ld` can use MRI compatible linker scripts as an alternative to the more general-purpose linker scripting language described in Chapter 3 "Command Language," page 15. MRI compatible linker scripts have a much simpler command set than the scripting language otherwise used with `ld`. GNU `ld` supports the most commonly used MRI linker commands; these commands are described here.

In general, MRI scripts aren't of much use with the `a.out` object file format, since it only has three sections and MRI scripts lack some features to make use of them.

You can specify a file containing an MRI-compatible script using the '`-c`' command-line option.

Each command in an MRI-compatible script occupies its own line; each command line starts with the keyword that identifies the command (though blank lines are also allowed for punctuation). If a line of an MRI-compatible script begins with an unrecognized keyword, `ld` issues a warning message, but continues processing the script.

Lines beginning with '`*`' are comments.

You can write these commands using all upper-case letters, or all lower case; for example, '`chip`' is the same as '`CHIP`'. The following list shows only the upper-case form of each command.

`ABSOLUTE` *secname*

`ABSOLUTE` *secname, secname, . . . secname*

Normally, `ld` includes in the output file all sections from all the input files. However, in an MRI-compatible script, you can use the `ABSOLUTE` command to restrict the sections that will be present in your output program. If the `ABSOLUTE` command is used at all in a script, then only the sections named explicitly in `ABSOLUTE` commands will appear in the linker output. You can still use other input sections (whatever you select on the command line, or using `LOAD`) to resolve addresses in the output file.

`ALIAS` *out-secname, in-secname*

Use this command to place the data from input section *in-secname* in a section called *out-secname* in the linker output file.

in-secname may be an integer.

`BASE` *expression*

Use the value of *expression* as the lowest address (other than absolute addresses) in the output file.

`CHIP` *expression*

`CHIP` *expression*, *expression*

This command does nothing; it is accepted only for compatibility.

`END` This command does nothing whatever; it's only accepted for compatibility.

`FORMAT` *output-format*

Similar to the `OUTPUT_FORMAT` command in the more general linker language, but restricted to one of these output formats:

1. S-records, if *output-format* is 'S'
2. IEEE, if *output-format* is 'IEEE'
3. COFF (the 'coff-m68k' variant in BFD), if *output-format* is 'COFF'

`LIST` *anything*. . .

Print (to the standard output file) a link map, as produced by the `ld` command-line option '-M'.

The keyword `LIST` may be followed by anything on the same line, with no change in its effect.

`LOAD` *filename*

`LOAD` *filename*, *filename*, . . . *filename*

Include one or more object file *filename* in the link; this has the same effect as specifying *filename* directly on the `ld` command line.

`NAME` *output-name*

output-name is the name for the program produced by `ld`; the MRI-compatible command `NAME` is equivalent to the command-line option '-o' or the general script language command `OUTPUT`.

`ORDER` *secname*, *secname*, . . . *secname*

`ORDER` *secname secname secname*

Normally, `ld` orders the sections in its output file in the order in which they first appear in the input files. In an MRI-compatible script, you can override this ordering with the `ORDER` command. The sections you list with `ORDER` will appear first in your output file, in the order specified.

`PUBLIC` *name*=*expression*

`PUBLIC` *name*,*expression*

`PUBLIC` *name expression*

Supply a value (*expression*) for external symbol *name* used in the linker input files.

`SECT` *secname*`,` *expression*
`SECT` *secname*`=`*expression*
`SECT` *secname* *expression*

You can use any of these three forms of the `SECT` command to specify the start address (*expression*) for section *secname*. If you have more than one `SECT` statement for the same *secname*, only the *first* sets the start address.

Index

*

*(COMMON) 26
*(*section*) 25

-

-(.................................... 13
--verbose 11
--whole-archive 13
-A*arch* 4
-b *format* 4
-Bstatic 5
-c *MRI-cmdfile* 5
-d 5
-dc 5
-defsym *symbol*=*exp* 5
-dp 5
-dynamic-linker *file* 6
-e *entry* 6
-embedded-relocs 6
-F 6
-format 6
-g 6
-G 6
-help 6
-i 6
-l*archive* 6
-L*dir* 7
-M 7
-m *emulation* 7
-Map 7
-n 7
-N 7
-no-keep-memory 7
-noinhibit-exec 7
-o *output* 8
-oformat 8
-r 9
-R *file* 8
-relax 8
-relax on i960 36
-rpath 9
-s 9
-S 9
-soname 9
-t 10
-T *script* 10
-Tbss *org* 10
-Tdata *org* 10
-traditional-format 10
-Ttext *org* 10
-u *symbol* 10
-Ur 11
-v 11
-V 11
-version 11
-warn-comon 11
-warn-once 12
-x 13
-X 13
-y *symbol* 13

.

. ... 17

;

; 19

=

=*fill* 31

[

[*section*. . .], not supported 26

"

" 16

>

>*region* 31

0

0x 16

A

ABSOLUTE (MRI) 41

absolute and relocatable symbols 19
ABSOLUTE(*exp*) 20
ADDR(*section*) 20
ALIAS (MRI) 41
ALIGN(*exp*) 21
aligning sections 30
allocating memory 22
architectures 4
archive files, from cmd line 6
arithmetic 15
arithmetic operators 18
assignment in scripts 18
assignment, in section defn 28
AT (*ldadr*) 30

B

back end 37
BASE (MRI) 41
BFD canonical format 39
BFD requirements 37
binary input files 33
binary input format 4
BLOCK(*align*) 30
BYTE(*expression*) 29

C

C++ constructors, arranging in link ... 32
CHIP (MRI) 41
combining symbols, warnings on 11
command files 15
command line 3
commands, fundamental 15
comments 15
common allocation 5, 33
commons in output 26
compatibility, MRI 5
constructors 11
CONSTRUCTORS 32
constructors, arranging in link 32
contents of a section 25
CREATE_OBJECT_SYMBOLS 27
current output location 17

D

dbx 10
decimal integers 16
default input format 13
DEFINED(*symbol*) 21
deleting local symbols 13
direct output 29
discontinuous memory 22
dot 17
dynamic linker, from command line 6

E

emulation 7
END (MRI) 42
entry point, defaults 32
entry point, from command line 6
ENTRY(*symbol*) 31
expression evaluation order 18
expression syntax 15
expression, absolute 20
expressions in a section 27

F

filename 25
filename symbols 27
filename(*section*) 25
files and sections, section defn 25
files, including in output sections 25
fill pattern, entire section 31
FILL(*expression*) 29
first input file 34
first instruction 31
FLOAT 33
FORCE_COMMON_ALLOCATION 33
FORMAT (MRI) 42
format, output file 33
functions in expression language 20
fundamental script commands 15

G

GNU linker 1
GNUTARGET 13, 34
GROUP (*files*) 33
grouping input files 33
groups of archives 13

H

H8/300 support 35
header size 22
help 6
hexadecimal integers 16

holes 17
holes, filling 29

I

i960 support 35
including an entire archive 13
incremental link 6
INPUT (*files*) 33
input file format 34
input filename symbols 27
input files, displaying 10
input files, section defn 25
input format 4
input sections to output section 25
integer notation 16
integer suffixes 16
internal object-file format 39

K

K and M integer suffixes 16

L

l = ... 23
L, deleting symbols beginning 13
layout of output file 15
lazy evaluation 18
len = ... 23
LENGTH = ... 23
link map 7
LIST (MRI) 42
LOAD (MRI) 42
load address, specifying 30
loading, preventing 30
local symbols, deleting 13
location counter 17
LONG(*expression*) 29

M

M and K integer suffixes 16
machine architecture, output 33
machine dependencies 35
MEMORY 22
memory region attributes 23
memory regions and sections 31
memory usage 7
MIPS embedded PIC code 6
MRI compatibility 41

N

NAME (MRI) 42
names 16
naming memory regions 23
naming output sections 24
naming the output file 8, 33
negative integers 16
NEXT(*exp*) 22
NMAGIC 7
NOFLOAT 33
NOLOAD 30
Non constant expression 20

O

o = ... 23
objdump -i 37
object file management 37
object files 3
object formats available 37
object size 6
octal integers 16
OMAGIC 7
opening object files 37
Operators for arithmetic 18
options 3
ORDER (MRI) 42
org = ... 23
ORIGIN = ... 23
OUTPUT (*filename*) 33
output file after errors 7
output file layout 15
OUTPUT_ARCH (*bfdname*) 33
OUTPUT_FORMAT (*bfdname*) 33

P

partial link 9
path for libraries 34
precedence in expressions 18
prevent unnecessary loading 30
provide 20
PUBLIC (MRI) 42

Q

QUAD(*expression*) 29
quoted symbol names 16

R

read-only text 7
read/write from cmd line 7
regions of memory 22
relaxing addressing modes 8
relaxing on H8/300.................... 35
relaxing on i960....................... 36
relocatable and absolute symbols 19
relocatable output...................... 9
requirements for BFD 37
retaining specified symbols............. 8
rounding up location counter.......... 21
runtime library name 9
runtime library search path............ 9

S

scaled integers 16
script files............................. 10
search directory, from cmd line......... 7
search path, libraries 34
`SEARCH_DIR ( path )` 34
`SECT` (MRI) 42
section address 20, 30
section alignment 30
section definition...................... 24
section defn, full syntax............... 29
section fill pattern 31
section size............................ 22
section start 30
section, assigning to memory region .. 31
`SECTIONS`............................ 23
segment origins, cmd line............. 10
semicolon 19
`SHORT(expression)` 29
`SIZEOF(section)` 22
`sizeof_headers` 22
`SIZEOF_HEADERS` 22
specify load address................... 30
`split` 9, 10
standard Unix system.................. 3
start address, section 30
start of execution 31
`STARTUP ( filename )` 34
strip all symbols 9
strip debugger symbols................. 9
stripping all but some symbols 8
suffixes for integers 16
`symbol = expression ;` 28
symbol defaults 21
symbol definition, scripts 18
`symbol f= expression ;` 28
symbol names.......................... 16
symbol tracing 13
symbol-only input 8
symbols, from command line 5
symbols, relocatable and absolute..... 19
symbols, retaining selectively 8
synthesizing linker..................... 8
synthesizing on H8/300 35

T

`TARGET ( format )` 34
traditional format..................... 10

U

unallocated address, next 22
undefined symbol 10
undefined symbols, warnings on 12
uninitialized data 26
unspecified memory................... 29
usage.................................. 6

V

variables, defining 18
verbose 10
version................................ 11

W

warnings, on combining symbols...... 11
warnings, on undefined symbols 12
what is this? 1

The body of this manual is set in
pncr at 10.95pt,
with headings in **pncb at 10.95pt**
and examples in `pcrr`.
pncri at 10.95pt and
`pcrro`
are used for emphasis.

The GNU Binary Utilities

Version 2.2

May 1993

Roland H. Pesch
Jeffrey M. Osier
Cygnus Support

Cygnus Support
TeXinfo 2.122-95q3 (Cygnus)

Copyright © 1991, 1992, 1993, 1994 1995 Free Software Foundation, Inc.

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

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

Table of Contents

Introduction 1

1 ar 3
- 1.1 Controlling `ar` on the command line 4
- 1.2 Controlling `ar` with a script 6

2 ld 11

3 nm 13

4 objcopy 17

5 objdump 21

6 ranlib 25

7 size 27

8 strings 29

9 strip 31

10 c++filt 33

11 nlmconv 35

12 Selecting the target system 37
- 12.1 Target Selection 37
- 12.2 Architecture selection 39
- 12.3 Linker emulation selection 39

Index 41

Introduction

This brief manual contains preliminary documentation for the GNU binary utilities (collectively version 2.2):

`ar` Create, modify, and extract from archives

`nm` List symbols from object files

`objcopy` Copy and translate object files

`objdump` Display information from object files

`ranlib` Generate index to archive contents

`size` List file section sizes and total size

`strings` List printable strings from files

`strip` Discard symbols

`c++filt` Demangle encoded C++ symbols

`nlmconv` Convert object code into a Netware Loadable Module

1 ar

```
ar [-]p[mod [relpos]] archive [member...]
ar -M [ <mri-script ]
```

The GNU `ar` program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called *members* of the archive).

The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

GNU `ar` can maintain archives whose members have names of any length; however, depending on how `ar` is configured on your system, a limit on member-name length may be imposed for compatibility with archive formats maintained with other tools. If it exists, the limit is often 15 characters (typical of formats related to a.out) or 16 characters (typical of formats related to coff).

`ar` is considered a binary utility because archives of this sort are most often used as *libraries* holding commonly needed subroutines.

`ar` creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier '`s`'. Once created, this index is updated in the archive whenever `ar` makes a change to its contents (save for the '`q`' update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use '`nm -s`' or '`nm --print-armap`' to list this index table. If an archive lacks the table, another form of `ar` called `ranlib` can be used to add just the table.

GNU `ar` is designed to be compatible with two different facilities. You can control its activity using command-line options, like the different varieties of `ar` on Unix systems; or, if you specify the single command-line option '`-M`', you can control it with a script supplied via standard input, like the MRI "librarian" program.

1.1 Controlling `ar` on the command line

```
ar [-]p[mod [relpos]] archive [member...]
```

When you use `ar` in the Unix style, `ar` insists on at least two arguments to execute: one keyletter specifying the *operation* (optionally accompanied by other keyletters specifying *modifiers*), and the archive name to act on.

Most operations can also accept further *member* arguments, specifying particular files to operate on.

GNU `ar` allows you to mix the operation code *p* and modifier flags *mod* in any order, within the first command-line argument.

If you wish, you may begin the first command-line argument with a dash.

The *p* keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

`d` *Delete* modules from the archive. Specify the names of modules to be deleted as *member*...; the archive is untouched if you specify no files to delete.

If you specify the '`v`' modifier, `ar` lists each module as it is deleted.

`m` Use this operation to *move* members in an archive.

The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

If no modifiers are used with `m`, any members you name in the *member* arguments are moved to the *end* of the archive; you can use the '`a`', '`b`', or '`i`' modifiers to move them to a specified place instead.

`p` *Print* the specified members of the archive, to the standard output file. If the '`v`' modifier is specified, show the member name before copying its contents to standard output.

If you specify no *member* arguments, all the files in the archive are printed.

`q` *Quick append*; add the files *member*... to the end of *archive*, without checking for replacement.

The modifiers '`a`', '`b`', and '`i`' do *not* affect this operation; new members are always placed at the end of the archive.

The modifier '`v`' makes `ar` list each file as it is appended.

Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can

use 'ar s' or `ranlib` explicitly to update the symbol table index.

`r` Insert the files *member*... into *archive* (with *replacement*). This operation differs from 'q' in that any previously existing members are deleted if their names match those being added.

If one of the files named in *member*... does not exist, `ar` displays an error message, and leaves undisturbed any existing members of the archive matching that name.

By default, new members are added at the end of the file; but you may use one of the modifiers 'a', 'b', or 'i' to request placement relative to some existing member.

The modifier 'v' used with this operation elicits a line of output for each file inserted, along with one of the letters 'a' or 'r' to indicate whether the file was appended (no old member deleted) or replaced.

`t` Display a *table* listing the contents of *archive*, or those of the files listed in *member*... that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the 'v' modifier.

If you do not specify a *member*, all files in the archive are listed.

If there is more than one file with the same name (say, 'fie') in an archive (say 'b.a'), 'ar t b.a fie' lists only the first instance; to see them all, you must ask for a complete listing—in our example, 'ar t b.a'.

`x` *Extract* members (named *member*) from the archive. You can use the 'v' modifier with this operation, to request that `ar` list each name as it extracts it.

If you do not specify a *member*, all files in the archive are extracted.

A number of modifiers (*mod*) may immediately follow the *p* keyletter, to specify variations on an operation's behavior:

`a` Add new files *after* an existing member of the archive. If you use the modifier 'a', the name of an existing archive member must be present as the *relpos* argument, before the *archive* specification.

`b` Add new files *before* an existing member of the archive. If you use the modifier 'b', the name of an existing archive member must be present as the *relpos* argument, before the *archive* specification. (same as 'i').

`c` *Create* the archive. The specified *`archive`* is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

`i` Insert new files *before* an existing member of the archive. If you use the modifier '`i`', the name of an existing archive member must be present as the *`relpos`* argument, before the *`archive`* specification. (same as '`b`').

`l` This modifier is accepted but not used.

`o` Preserve the *original* dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

`s` Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running '`ar s`' on an archive is equivalent to running '`ranlib`' on it.

`u` Normally, '`ar r`'... inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The '`u`' modifier is allowed only for the operation '`r`' (replace). In particular, the combination '`qu`' is not allowed, since checking the timestamps would lose any speed advantage from the operation '`q`'.

`v` This modifier requests the *verbose* version of an operation. Many operations display additional information, such as filenames processed, when the modifier '`v`' is appended.

`V` This modifier shows the version number of `ar`.

1.2 Controlling `ar` with a script

```
ar -M [ <script ]
```

If you use the single command-line option '`-M`' with `ar`, you can control its operation with a rudimentary command language. This form of `ar` operates interactively if standard input is coming directly from a terminal. During interactive use, `ar` prompts for input (the prompt is '`AR >`'), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and `ar` abandons execution (with a nonzero exit code) on any error.

The `ar` command language is *not* designed to be equivalent to the command-line options; in fact, it provides somewhat less control over

archives. The only purpose of the command language is to ease the transition to GNU `ar` for developers who already have scripts written for the MRI "librarian" program.

The syntax for the `ar` command language is straightforward:

- commands are recognized in upper or lower case; for example, `LIST` is the same as `list`. In the following descriptions, commands are shown in upper case for clarity.
- a single command may appear on each line; it is the first word on the line.
- empty lines are allowed, and have no effect.
- comments are allowed; text after either of the characters '`*`' or '`;`' is ignored.
- Whenever you use a list of names as part of the argument to an `ar` command, you can separate the individual names with either commas or blanks. Commas are shown in the explanations below, for clarity.
- '`+`' is used as a line continuation character; if '`+`' appears at the end of a line, the text on the following line is considered part of the current command.

Here are the commands you can use in `ar` scripts, or when using `ar` interactively. Three of them have special significance:

`OPEN` or `CREATE` specify a *current archive*, which is a temporary file required for most of the other commands.

`SAVE` commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.

`ADDLIB` *`archive`*
`ADDLIB` *`archive`* `(`*`module`*`,` *`module`*`, . . .` *`module`*`)`
: Add all the contents of *`archive`* (or, if specified, each named *`module`* from *`archive`*) to the current archive.

  Requires prior use of `OPEN` or `CREATE`.

`ADDMOD` *`member`*`,` *`member`*`, . . .` *`member`*
: Add each named *`member`* as a module in the current archive.

  Requires prior use of `OPEN` or `CREATE`.

`CLEAR`
: Discard the contents of the current archive, cancelling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.

`CREATE` *`archive`*
: Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as

archive until you use SAVE. You can overwrite existing archives; similarly, the contents of any existing file named *archive* will not be destroyed until SAVE.

DELETE *module, module, . . . module*
: Delete each listed *module* from the current archive; equivalent to 'ar -d *archive module . . . module*'.

  Requires prior use of OPEN or CREATE.

DIRECTORY *archive* (*module, . . . module*)
DIRECTORY *archive* (*module, . . . module*) *outputfile*
: List each named *module* present in *archive*. The separate command VERBOSE specifies the form of the output: when verbose output is off, output is like that of 'ar -t *archive module*. . .'. When verbose output is on, the listing is like 'ar -tv *archive module*. . .'.

  Output normally goes to the standard output stream; however, if you specify *outputfile* as a final argument, ar directs the output to that file.

END
: Exit from ar, with a 0 exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last SAVE command, those changes are lost.

EXTRACT *module, module, . . . module*
: Extract each named *module* from the current archive, writing them into the current directory as separate files. Equivalent to 'ar -x *archive module*. . .'.

  Requires prior use of OPEN or CREATE.

LIST
: Display full contents of the current archive, in "verbose" style regardless of the state of VERBOSE. The effect is like 'ar tv *archive*'). (This single command is a GNU ld enhancement, rather than present for MRI compatibility.)

  Requires prior use of OPEN or CREATE.

OPEN *archive*
: Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect *archive* until you next use SAVE.

REPLACE *module, module, . . . module*
: In the current archive, replace each existing *module* (named in the REPLACE arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist.

Requires prior use of `OPEN` or `CREATE`.

`VERBOSE`
: Toggle an internal flag governing the output from `DIRECTORY`. When the flag is on, `DIRECTORY` output matches output from '`ar -tv`'....

`SAVE`
: Commit your changes to the current archive, and actually save it as a file with the name specified in the last `CREATE` or `OPEN` command.

  Requires prior use of `OPEN` or `CREATE`.

2 ld

The GNU linker `ld` is now described in a separate manual. See section "Overview" in *Using LD: the GNU linker*.

3 nm

```
nm [ -a | --debug-syms ]  [ -g | --extern-only ]
   [ -B ]  [ -C | --demangle ] [ -D | --dynamic ]
   [ -s | --print-armap ]  [ -A | -o | --print-file-name ]
   [ -n | -v | --numeric-sort ]  [ -p | --no-sort ]
   [ -r | --reverse-sort ]  [ --size-sort ] [ -u | --undefined-
only ]
   [ -t radix | --radix=radix ] [ -P | --portability ]
   [ --target=bfdname ] [ -f format | --format=format ]
   [ --no-demangle ] [ -V | --version ]  [ --help ]
                          [ objfile. . . ]
```

GNU `nm` lists the symbols from object files *objfile*.... If no object files are listed as arguments, `nm` assumes 'a.out'.

For each symbol, `nm` shows:

- The symbol value, in the radix selected by options (see below), or hexadecimal by default.
- The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

  `A` Absolute.

  `B` BSS (uninitialized data).

  `C` Common.

  `D` Initialized data.

  `I` Indirect reference.

  `T` Text (program code).

  `U` Undefined.
- The symbol name.

The long and short forms of options, shown here as alternatives, are equivalent.

`-A`
`-o`
`--print-file-name`
: Precede each symbol by the name of the input file (or archive element) in which it was found, rather than identifying the input file once only, before all of its symbols.

`-a`
`--debug-syms`
: Display all symbols, even debugger-only symbols; normally these are not listed.

`-B` The same as '`--format=bsd`' (for compatibility with the MIPS `nm`).

`-C`
`--demangle`
Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See Chapter 10 "c++filt," page 33, for more information on demangling.

`--no-demangle`
Do not demangle low-level symbol names. This is the default.

`-D`
`--dynamic`
Display the dynamic symbols rather than the normal symbols. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`-f` *`format`*
`--format=`*`format`*
Use the output format *`format`*, which can be `bsd`, `sysv`, or `posix`. The default is `bsd`. Only the first character of *`format`* is significant; it can be either upper or lower case.

`-g`
`--extern-only`
Display only external symbols.

`-n`
`-v`
`--numeric-sort`
Sort symbols numerically by their addresses, rather than alphabetically by their names.

`-p`
`--no-sort`
Do not bother to sort the symbols in any order; print them in the order encountered.

`-P`
`--portability`
Use the POSIX.2 standard output format instead of the default format. Equivalent to '`-f posix`'.

`-s`
`--print-armap`
When listing symbols from archive members, include the index: a mapping (stored in the archive by `ar` or `ranlib`) of which modules contain definitions for which names.

`-r`
`--reverse-sort`
: Reverse the order of the sort (whether numeric or alphabetic); let the last come first.

`--size-sort`
: Sort symbols by size. The size is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.

`-t` *radix*
`--radix=`*radix*
: Use *radix* as the radix for printing the symbol values. It must be '`d`' for decimal, '`o`' for octal, or '`x`' for hexadecimal.

`--target=`*bfdname*
: Specify an object code format other than your system's default format. See Section 12.1 "Target Selection," page 37, for more information.

`-u`
`--undefined-only`
: Display only undefined symbols (those external to each object file).

`-V`
`--version`
: Show the version number of `nm` and exit.

`--help`
: Show a summary of the options to `nm` and exit.

4 objcopy

```
objcopy [ -F bfdname | --target=bfdname ]
        [ -I bfdname | --input-target=bfdname ]
        [ -O bfdname | --output-target=bfdname ]
        [ -S | --strip-all ]  [ -g | --strip-debug ]
        [ -K symbolname | --keep-symbol=symbolname ]
        [ -N symbolname | --strip-symbol=symbolname ]
        [ -x | --discard-all ]  [ -X | --discard-locals ]
        [ -b byte | --byte=byte ]
        [ -i interleave | --interleave=interleave ]
        [ -R sectionname | --remove-section=sectionname ]
        [ --gap-fill=val ] [ --pad-to=address ]
        [ --set-start=val ] [ --adjust-start=incr ]
        [ --adjust-vma=incr ]
        [ --adjust-section-vma=section{=,+,-}val ]
        [ --adjust-warnings ] [ --no-adjust-warnings ]
        [ --set-section-flags=section=flags ]
        [ --add-section=sectionname=filename ]
        [ -v | --verbose ] [ -V | --version ]  [ --help ]
        infile [outfile]
```

The GNU `objcopy` utility copies the contents of an object file to another. `objcopy` uses the GNU BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of `objcopy` is controlled by command-line options.

`objcopy` creates temporary files to do its translations and deletes them afterward. `objcopy` uses BFD to do all its translation work; it has access to all the formats described in BFD and thus is able to recognize most formats without being told explicitly. See section “BFD” in *Using LD*.

`objcopy` can be used to generate S-records by using an output target of '`srec`' (e.g., use '`-O srec`').

`objcopy` can be used to generate a raw binary file by using an output target of '`binary`' (e.g., use '`-O binary`'). When `objcopy` generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the virtual address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use '`-S`' to remove sections containing debugging information. In some cases '`-R`' will be useful to remove sections which contain information which is not needed by the binary file.

infile
outfile The source and output files, respectively. If you do not specify *outfile*, objcopy creates a temporary file and destructively renames the result with the name of *infile*.

-I *bfdname*
--input-target=*bfdname*
Consider the source file's object format to be *bfdname*, rather than attempting to deduce it. See Section 12.1 "Target Selection," page 37, for more information.

-O *bfdname*
--output-target=*bfdname*
Write the output file using the object format *bfdname*. See Section 12.1 "Target Selection," page 37, for more information.

-F *bfdname*
--target=*bfdname*
Use *bfdname* as the object format for both the input and the output file; i.e., simply transfer data from source to destination with no translation. See Section 12.1 "Target Selection," page 37, for more information.

-R *sectionname*
--remove-section=*sectionname*
Remove any section named *sectionname* from the output file. This option may be given more than once. Using this option inappropriately may make the output file unusable.

-S
--strip-all
Do not copy relocation and symbol information from the source file.

-g
--strip-debug
Do not copy debugging symbols from the source file.

-K *symbolname*
--keep-symbol=*symbolname*
Copy only symbol *symbolname* from the source file. This option may be given more than once.

-N *symbolname*
--strip-symbol=*symbolname*
Do not copy symbol *symbolname* from the source file. This option may be given more than once, and may be combined with strip options other than -K.

`-x`
`--discard-all`
Do not copy non-global symbols from the source file.

`-X`
`--discard-locals`
Do not copy compiler-generated local symbols. (These usually start with '`L`' or '`.`'.)

`-b` *byte*
`--byte=`*byte*
Keep only every *byte*th byte of the input file (header data is not affected). *byte* can be in the range from 0 to *interleave*-1, where *interleave* is given by the '`-i`' or '`--interleave`' option, or the default of 4. This option is useful for creating files to program ROM. It is typically used with an `srec` output target.

`-i` *interleave*
`--interleave=`*interleave*
Only copy one out of every *interleave* bytes. Select which byte to copy with the *-b* or '`--byte`' option. The default is 4. `objcopy` ignores this option if you do not specify either '`-b`' or '`--byte`'.

`--gap-fill` *val*
Fill gaps between sections with *val*. This is done by increasing the size of the section with the lower address, and filling in the extra space created with *val*.

`--pad-to` *address*
Pad the output file up to the virtual address *address*. This is done by increasing the size of the last section. The extra space is filled in with the value specified by '`--gap-fill`' (default zero).

`--set-start` *val*
Set the address of the new file to *val*. Not all object file formats support setting the start address.

`--adjust-start` *incr*
Adjust the start address by adding *incr*. Not all object file formats support setting the start address.

`--adjust-vma` *incr*
Adjust the address of all sections, as well as the start address, by adding *incr*. Some object file formats do not permit section addresses to be changed arbitrarily. Note that this does not relocate the sections; if the program expects sections to

be loaded at a certain address, and this option is used to change the sections such that they are loaded at a different address, the program may fail.

`--adjust-section-vma` *`section`*`{=,+,-}`*`val`*
: Set or adjust the address of the named *`section`*. If '`=`' is used, the section address is set to *`val`*. Otherwise, *`val`* is added to or subtracted from the section address. See the comments under '`--adjust-vma`', above. If *`section`* does not exist in the input file, a warning will be issued, unless '`--no-adjust-warnings`' is used.

`--adjust-warnings`
: If '`--adjust-section-vma`' is used, and the named section does not exist, issue a warning. This is the default.

`--no-adjust-warnings`
: Do not issue a warning if '`--adjust-section-vma`' is used, even if the named section does not exist.

`--set-section-flags` *`section=flags`*
: Set the flags for the named section. The *`flags`* argument is a comma separated string of flag names. The recognized names are '`alloc`', '`load`', '`readonly`', '`code`', '`data`', and '`rom`'. Not all flags are meaningful for all object file formats.

`--add-section` *`sectionname=filename`*
: Add a new section named *`sectionname`* while copying the file. The contents of the new section are taken from the file *`filename`*. The size of the section will be the size of the file. This option only works on file formats which can support sections with arbitrary names.

`-V`
`--version`
: Show the version number of `objcopy`.

`-v`
`--verbose`
: Verbose output: list all object files modified. In the case of archives, '`objcopy -V`' lists all members of the archive.

`--help`
: Show a summary of the options to `objcopy`.

5 objdump

```
objdump [ -a | --archive-headers ]
        [ -b bfdname | --target=bfdname ]
        [ -d | --disassemble ]  [ -D | --disassemble-all ]
        [ -f | --file-headers ]
        [ -h | --section-headers | --headers ]  [ -i | --info ]
        [ -j section | --section=section ]
        [ -l | --line-numbers ] [ -S | --source ]
        [ -m machine | --architecture=machine ]
        [ -r | --reloc ] [ -R | --dynamic-reloc ]
        [ -s | --full-contents ]  [ --stabs ]
        [ -t | --syms ] [ -T | --dynamic-syms ] [ -x |
               --all-headers ]
        [ --version ]  [ --help ] objfile...
```

`objdump` displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

objfile... are the object files to be examined. When you specify archives, `objdump` shows information on each of the member object files.

The long and short forms of options, shown here as alternatives, are equivalent. At least one option besides '`-l`' must be given.

`-a`
`--archive-header`
: If any of the *objfile* files are archives, display the archive header information (in a format similar to '`ls -l`'). Besides the information you could list with '`ar tv`', '`objdump -a`' shows the object file format of each archive member.

`-b` *bfdname*
`--target=`*bfdname*
: Specify that the object-code format for the object files is *bfdname*. This option may not be necessary; *objdump* can automatically recognize many formats.

For example,

```
objdump -b oasys -m vax -h fu.o
```

displays summary information from the section headers ('`-h`') of '`fu.o`', which is explicitly identified ('`-m`') as a VAX object file in the format produced by Oasys compilers. You can list the formats available with the '`-i`' option. See Section 12.1 “Target Selection,” page 37, for more information.

`-d`
`--disassemble`
: Display the assembler mnemonics for the machine instructions from *`objfile`*. This option only disassembles those sections which are expected to contain instructions.

`-D`
`--disassemble-all`
: Like '`-d`', but disassemble the contents of all sections, not just those expected to contain instructions.

`-f`
`--file-header`
: Display summary information from the overall header of each of the *`objfile`* files.

`-h`
`--section-header`
`--header`
: Display summary information from the section headers of the object file.

  File segments may be relocated to nonstandard addresses, for example by using the '`-Ttext`', '`-Tdata`', or '`-Tbss`' options to `ld`. However, some object file formats, such as a.out, do not store the starting address of the file segments. In those situations, although `ld` relocates the sections correctly, using '`objdump -h`' to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.

`--help`
: Print a summary of the options to `objdump` and exit.

`-i`
`--info`
: Display a list showing all architectures and object formats available for specification with '`-b`' or '`-m`'.

`-j` *`name`*
`--section=`*`name`*
: Display information only for section *`name`*.

`-l`
`--line-numbers`
: Label the display (using debugging information) with the filename and source line numbers corresponding to the object code shown. Only useful with '`-d`' or '`-D`'.

`-m` *`machine`*
`--architecture=`*`machine`*
: Specify that the object files *`objfile`* are for architecture *`machine`*. List available architectures using the '`-i`' option.

`-r`
`--reloc` Print the relocation entries of the file. If used with '`-d`' or '`-D`', the relocations are printed interspersed with the disassembly.

`-R`
`--dynamic-reloc`
Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`-s`
`--full-contents`
Display the full contents of any sections requested.

`-S`
`--source` Display source code intermixed with disassembly, if possible. Implies '`-d`'.

`--stabs` Display the full contents of any sections requested. Display the contents of the .stab and .stab.index and .stab.excl sections from an ELF file. This is only useful on systems (such as Solaris 2.0) in which `.stab` debugging symbol-table entries are carried in an ELF section. In most other file formats, debugging symbol-table entries are interleaved with linkage symbols, and are visible in the '`--syms`' output.

`-t`
`--syms` Print the symbol table entries of the file. This is similar to the information provided by the '`nm`' program.

`-T`
`--dynamic-syms`
Print the dynamic symbol table entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries. This is similar to the information provided by the '`nm`' program when given the '`-D`' ('`--dynamic`') option.

`--version`
Print the version number of `objdump` and exit.

`-x`
`--all-header`
Display all available header information, including the symbol table and relocation entries. Using '`-x`' is equivalent to specifying all of '`-a -f -h -r -t`'.

6 ranlib

```
ranlib [-vV] archive
```

`ranlib` generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file.

You may use '`nm -s`' or '`nm --print-armap`' to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

The GNU `ranlib` program is another form of GNU `ar`; running `ranlib` is completely equivalent to executing '`ar -s`'. See Chapter 1 "ar," page 3.

`-v`
`-V` Show the version number of `ranlib`.

7 size

```
size [ -A | -B | --format=compatibility ]
     [ --help ]  [ -d | -o | -x | --radix=number ]
     [ --target=bfdname ]  [ -V | --version ]
     objfile...
```

The GNU `size` utility lists the section sizes—and the total size—for each of the object or archive files *objfile* in its argument list. By default, one line of output is generated for each object file or each module in an archive.

objfile... are the object files to be examined.

The command line options have the following meanings:

`-A`
`-B`
`--format=`*compatibility*

Using one of these options, you can choose whether the output from GNU `size` resembles output from System V `size` (using '`-A`', or '`--format=sysv`'), or Berkeley `size` (using '`-B`', or '`--format=berkeley`'). The default is the one-line format similar to Berkeley's.

Here is an example of the Berkeley (default) format of output from `size`:

```
size --format=Berkeley ranlib size
text    data    bss     dec     hex     filename
294880  81920   11592   388392  5ed28   ranlib
294880  81920   11888   388688  5ee50   size
```

This is the same data, but displayed closer to System V conventions:

```
size --format=SysV ranlib size
ranlib  :
section         size         addr
.text         294880         8192
.data          81920       303104
.bss           11592       385024
Total         388392


size  :
section         size         addr
.text         294880         8192
.data          81920       303104
.bss           11888       385024
Total         388688
```

`--help` Show a summary of acceptable arguments and options.

`-d`
`-o`
`-x`
`--radix=`*number*

Using one of these options, you can control whether the size of each section is given in decimal ('`-d`', or '`--radix=10`'); octal ('`-o`', or '`--radix=8`'); or hexadecimal ('`-x`', or '`--radix=16`'). In '`--radix=`*number*', only the three values (8, 10, 16) are supported. The total size is always given in two radices; decimal and hexadecimal for '`-d`' or '`-x`' output, or octal and hexadecimal if you're using '`-o`'.

`--target=`*bfdname*

Specify that the object-code format for *objfile* is *bfdname*. This option may not be necessary; `size` can automatically recognize many formats. See Section 12.1 "Target Selection," page 37, for more information.

`-V`
`--version`

Display the version number of `size`.

8 strings

```
strings [-afov] [-min-len] [-n min-len] [-t radix] [-]
        [--all] [--print-file-name] [--bytes=min-len]
        [--radix=radix] [--target=bfdname]
        [--help] [--version] file...
```

For each *file* given, GNU `strings` prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by a NUL or newline character. By default, it only prints the strings from the initialized data sections of object files; for other types of files, it prints the strings from the whole file.

`strings` is mainly useful for determining the contents of non-text files.

`-a`
`--all`
`-` Do not scan only the initialized data section of object files; scan the whole files.

`-f`
`--print-file-name`
Print the name of the file before each string.

`--help` Print a summary of the program usage on the standard output and exit.

`-`*min-len*

`-n` *min-len*
`--bytes=`*min-len*
Print sequences of characters that are at least *min-len* characters long, instead of the default 4.

`-o` Like '`-t o`'. Some other versions of `strings` have '`-o`' act like '`-t d`' instead. Since we can not be compatible with both ways, we simply chose one.

`-t` *radix*
`--radix=`*radix*
Print the offset within the file before each string. The single character argument specifies the radix of the offset—'`o`' for octal, '`x`' for hexadecimal, or '`d`' for decimal.

`--target=`*bfdname*
Specify an object code format other than your system's default format. See Section 12.1 “Target Selection,” page 37, for more information.

`-v`
`--version`
Print the program version number on the standard output and exit.

9 strip

```
strip [ -F bfdname | --target=bfdname | --target=bfdname ]
      [ -I bfdname | --input-target=bfdname ]
      [ -O bfdname | --output-target=bfdname ]
      [ -s | --strip-all ] [ -S | -g | --strip-debug ]
      [ -K symbolname | --keep-symbol=symbolname ]
      [ -N symbolname | --strip-symbol=symbolname ]
      [ -x | --discard-all ] [ -X | --discard-locals ]
      [ -R sectionname | --remove-section=sectionname ]
      [ -v | --verbose ]  [ -V | --version ]  [ --help ]
      objfile. . .
```

GNU `strip` discards all symbols from object files *objfile*. The list of object files may include archives. At least one object file must be given.

`strip` modifies the files named in its argument, rather than writing modified copies under different names.

`-F` *bfdname*
`--target=`*bfdname*
: Treat the original *objfile* as a file with the object code format *bfdname*, and rewrite it in the same format. See Section 12.1 "Target Selection," page 37, for more information.

`--help`
: Show a summary of the options to `strip` and exit.

`-I` *bfdname*
`--input-target=`*bfdname*
: Treat the original *objfile* as a file with the object code format *bfdname*. See Section 12.1 "Target Selection," page 37, for more information.

`-O` *bfdname*
`--output-target=`*bfdname*
: Replace *objfile* with a file in the output format *bfdname*. See Section 12.1 "Target Selection," page 37, for more information.

`-R` *sectionname*
`--remove-section=`*sectionname*
: Remove any section named *sectionname* from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable.

`-s`
`--strip-all`
: Remove all symbols.

`-g`
`-S`
`--strip-debug`
: Remove debugging symbols only.

`-K` *`symbolname`*
`--keep-symbol=`*`symbolname`*
: Keep only symbol *`symbolname`* from the source file. This option may be given more than once.

`-N` *`symbolname`*
`--strip-symbol=`*`symbolname`*
: Remove symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with strip options other than `-K`.

`-x`
`--discard-all`
: Remove non-global symbols.

`-X`
`--discard-locals`
: Remove compiler-generated local symbols. (These usually start with ‘`L`’ or ‘`.`’.)

`-V`
`--version`
: Show the version number for `strip`.

`-v`
`--verbose`
: Verbose output: list all object files modified. In the case of archives, ‘`strip -v`’ lists all members of the archive.

10 c++filt

```
c++filt [ -_ | --strip-underscores ]
[ -n | --no-strip-underscores ]
        [ -s format | --format=format ]
        [ --help ]  [ --version ]  [ symbol... ]
```

The C++ language provides function overloading, which means that you can write many functions with the same name (providing each takes parameters of different types). All C++ function names are encoded into a low-level assembly label (this process is known as *mangling*). The `c++filt` program does the inverse mapping: it decodes (*demangles*) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

You can use `c++filt` to decipher individual symbols:

```
c++filt symbol
```

If no *symbol* arguments are given, `c++filt` reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

`-_`
`--strip-underscores`
: On some systems, both the C and C++ compilers put an underscore in front of every name. For example, the C name `foo` gets the low-level name `_foo`. This option removes the initial underscore. Whether `c++filt` removes the underscore by default is target dependent.

`-n`
`--no-strip-underscores`
: Do not remove the initial underscore.

`-s` *format*
`--format=`*format*
: GNU `nm` can decode three different methods of mangling, used by different C++ compilers. The argument to this option selects which method it uses:

  `gnu` the one used by the GNU compiler (the default method)

  `lucid` the one used by the Lucid compiler

  `arm` the one specified by the C++ Annotated Reference Manual

`--help` Print a summary of the options to `c++filt` and exit.

`--version`
Print the version number of `c++filt` and exit.

Warning: `c++filt` is a new utility, and the details of its user interface are subject to change in future releases. In particular, a command-line option may be required in the the future to decode a name passed as an argument on the command line; in other words,

```
c++filt symbol
```

may in a future release become

```
c++filt option symbol
```

11 nlmconv

`nlmconv` converts a relocatable object file into a NetWare Loadable Module.

> *Warning:* `nlmconv` is not always built as part of the binary utilities, since it is only useful for NLM targets.

```
nlmconv [ -I bfdname | --input-target=bfdname ]
        [ -O bfdname | --output-target=bfdname ]
        [ -T headerfile | --header-file=headerfile ]
        [ -d | --debug]  [ -l linker | --linker=linker ]
        [ -h | --help ]  [ -V | --version ]
        infile outfile
```

`nlmconv` converts the relocatable '`i386`' object file *`infile`* into the NetWare Loadable Module *`outfile`*, optionally reading *`headerfile`* for NLM header information. For instructions on writing the NLM command file language used in header files, see the '`linkers`' section, '`NLMLINK`' in particular, of the *NLM Development and Tools Overview*, which is part of the NLM Software Developer's Kit ("NLM SDK"), available from Novell, Inc. `nlmconv` uses the GNU Binary File Descriptor library to read *`infile`*; see section "BFD" in *Using LD*, for more information.

`nlmconv` can perform a link step. In other words, you can list more than one object file for input if you list them in the definitions file (rather than simply specifying one input file on the command line). In this case, `nlmconv` calls the linker for you.

`-I` *`bfdname`*
`--input-target=`*`bfdname`*
: Object format of the input file. `nlmconv` can usually determine the format of a given file (so no default is necessary). See Section 12.1 "Target Selection," page 37, for more information.

`-O` *`bfdname`*
`--output-target=`*`bfdname`*
: Object format of the output file. `nlmconv` infers the output format based on the input format, e.g. for a '`i386`' input file the output format is '`nlm32-i386`'. See Section 12.1 "Target Selection," page 37, for more information.

`-T` *`headerfile`*
`--header-file=`*`headerfile`*
: Reads *`headerfile`* for NLM header information. For instructions on writing the NLM command file language used in header files, see see the '`linkers`' section, of the *NLM Development and Tools Overview*, which is part of the NLM Software Developer's Kit, available from Novell, Inc.

`-d`
`--debug` Displays (on standard error) the linker command line used by `nlmconv`.

`-l` *`linker`*
`--linker=`*`linker`*
Use *`linker`* for any linking. *`linker`* can be an abosolute or a relative pathname.

`-h`
`--help` Prints a usage summary.

`-V`
`--version`
Prints the version number for `nlmconv`.

12 Selecting the target system

You can specify three aspects of the target system to the GNU binary file utilities, each in several ways:

- the target
- the architecture
- the linker emulation (which applies to the linker only)

In the following summaries, the lists of ways to specify values are in order of decreasing precedence. The ways listed first override those listed later.

The commands to list valid values only list the values for which the programs you are running were configured. If they were configured with '`--with-targets=all`', the commands list most of the available values, but a few are left out; not all targets can be configured in at once because some of them can only be configured *native* (on hosts with the same type as the target system).

12.1 Target Selection

A *target* is an object file format. A given target may be supported for multiple architectures (see Section 12.2 “Architecture Selection,” page 39). A target selection may also have variations for different operating systems or architectures.

The command to list valid target values is '`objdump -i`' (the first column of output contains the relevant information).

Some sample values are: '`a.out-hp300bsd`', '`ecoff-littlemips`', '`a.out-sunos-big`'.

`objdump` Target

Ways to specify:

1. command line option: '`-b`' or '`--target`'
2. environment variable `GNUTARGET`
3. deduced from the input file

`objcopy` and `strip` Input Target

Ways to specify:

1. command line options: '`-I`' or '`--input-target`', or '`-F`' or '`--target`'
2. environment variable `GNUTARGET`
3. deduced from the input file

objcopy and strip Output Target

Ways to specify:

1. command line options: '-O' or '--output-target', or '-F' or '--target'
2. the input target (see "objcopy and strip Input Target" above)
3. environment variable GNUTARGET
4. deduced from the input file

nm, size, and strings Target

Ways to specify:

1. command line option: '--target'
2. environment variable GNUTARGET
3. deduced from the input file

Linker Input Target

Ways to specify:

1. command line option: '-b' or '--format' (see section "Options" in *Using LD*)
2. script command TARGET (see section "Option Commands" in *Using LD*)
3. environment variable GNUTARGET (see section "Environment" in *Using LD*)
4. the default target of the selected linker emulation (see Section 12.3 "Linker Emulation Selection," page 39)

Linker Output Target

Ways to specify:

1. command line option: '-oformat' (see section "Options" in *Using LD*)
2. script command OUTPUT_FORMAT (see section "Option Commands" in *Using LD*)
3. the linker input target (see "Linker Input Target" above)

12.2 Architecture selection

An *architecture* is a type of CPU on which an object file is to run. Its name may contain a colon, separating the name of the processor family from the name of the particular CPU.

The command to list valid architecture values is 'objdump -i' (the second column contains the relevant information).

Sample values: 'm68k:68020', 'mips:3000', 'sparc'.

objdump Architecture

Ways to specify:

1. command line option: '-m' or '--architecture'
2. deduced from the input file

objcopy, nm, size, strings Architecture

Ways to specify:

1. deduced from the input file

Linker Input Architecture

Ways to specify:

1. deduced from the input file

Linker Output Architecture

Ways to specify:

1. script command OUTPUT_ARCH (see section "Option Commands" in *Using LD*)
2. the default architecture from the linker output target (see Section 12.1 "Target Selection," page 37)

12.3 Linker emulation selection

A linker *emulation* is a "personality" of the linker, which gives the linker default values for the other aspects of the target system. In particular, it consists of

- the linker script
- the target

- several "hook" functions that are run at certain stages of the linking process to do special things that some targets require

The command to list valid linker emulation values is '`ld -V`'.

Sample values: '`hp300bsd`', '`mipslit`', '`sun4`'.

Ways to specify:

1. command line option: '`-m`' (see section "Options" in *Using LD*)
2. environment variable `LDEMULATION`
3. compiled-in `DEFAULT_EMULATION` from '`Makefile`', which comes from `EMUL` in '`config/`*`target`*`.mt`'

Index

.

.stab 23

A

all header information, object file 23
ar 3
ar compatibility 3
architecture 22
architectures available 22
archive contents 25
archive headers 21
archives 3

C

c++filt 33
collections of files 3
compatibility, ar 3
contents of archive 5
creating archives 5

D

dates in archive 6
debug symbols 23
debugging symbols 13
deleting from archive 4
demangling C++ symbols 14, 33
disassembling object code 22
disassembly, with source 23
discarding symbols 31
dynamic relocation entries, in object file 23
dynamic symbol table entries, printing 23
dynamic symbols 14

E

ELF object file format 23
external symbols 14, 15
extract from archive 5

F

file name 13

H

header information, all 23

I

input file name 13

L

ld 11
libraries 3
linker 11
listings strings 29

M

machine instructions 22
moving in archive 4
MRI compatibility, ar 6

N

name duplication in archive 5
name length 3
nm 13
nm compatibility 13, 14
nm format 13, 14

O

objdump 21
object code format 15, 21, 28, 29
object file header 22
object file information 21
object file sections 23
object formats available 22
operations on archive 4

P

printing from archive 4
printing strings 29

Q

quick append to archive 4

R

radix for section sizes 28
ranlib 25
relative placement in archive 5
relocation entries, in object file 22
removing symbols 31
repeated names in archive 5
replacement in archive 5

S

scripts, `ar` 6
section headers 22
section information 22
section sizes 27
sections, full contents 23
size 27
`size` display format 27
`size` number format 28
sorting symbols 14
source disassembly 23
source file name 13
source filenames for object files 22
stab 23
strings 29
strings, printing 29
strip 31
symbol index 3, 25
symbol index, listing 14
symbol table entries, printing 23
symbols 13
symbols, discarding 31

U

undefined symbols 15
Unix compatibility, `ar` 4
updating an archive 6

V

version 1

W

writing archive index 6

GNU Make

A Program for Directing Recompilation
Edition 0.48, for `make` Version 3.73 Beta.
April 1995

Richard M. Stallman and Roland McGrath

Copyright © 1988, '89, '90, '91, '92, '93, '94, '95 Free Software Foundation, Inc.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA
Printed copies are available for $20 each.
ISBN 1-882114-50-7

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided 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 manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Cover art by Etienne Suvasa.

Short Contents

1 Overview of make 1
2 An Introduction to Makefiles 3
3 Writing Makefiles 11
4 Writing Rules 17
5 Writing the Commands in Rules 37
6 How to Use Variables 51
7 Conditional Parts of Makefiles 65
8 Functions for Transforming Text 71
9 How to Run make 83
10 Using Implicit Rules 95
11 Using make to Update Archive Files 115
12 Features of GNU make 119
13 Incompatibilities and Missing Features 123
14 Makefile Conventions 125
Appendix A Quick Reference 137
Appendix B Complex Makefile Example 143
Index of Concepts 149
Index of Functions, Variables, & Directives 157

Table of Contents

1 Overview of make 1

1.1 How to Read This Manual 1
1.2 Problems and Bugs 2

2 An Introduction to Makefiles 3

2.1 What a Rule Looks Like 3
2.2 A Simple Makefile 4
2.3 How make Processes a Makefile 5
2.4 Variables Make Makefiles Simpler 6
2.5 Letting make Deduce the Commands 7
2.6 Another Style of Makefile 8
2.7 Rules for Cleaning the Directory 9

3 Writing Makefiles 11

3.1 What Makefiles Contain 11
3.2 What Name to Give Your Makefile 12
3.3 Including Other Makefiles 12
3.4 The Variable MAKEFILES 14
3.5 How Makefiles Are Remade 14
3.6 Overriding Part of Another Makefile 16

4 Writing Rules 17

4.1 Rule Syntax 17
4.2 Using Wildcard Characters in File Names 18
4.2.1 Wildcard Examples 18
4.2.2 Pitfalls of Using Wildcards 19
4.2.3 The Function wildcard 20
4.3 Searching Directories for Dependencies 20
4.3.1 VPATH: Search Path for All Dependencies 21
4.3.2 The vpath Directive 21
4.3.3 Writing Shell Commands with Directory Search 23
4.3.4 Directory Search and Implicit Rules 23
4.3.5 Directory Search for Link Libraries 23
4.4 Phony Targets 24
4.5 Rules without Commands or Dependencies 26
4.6 Empty Target Files to Record Events 26
4.7 Special Built-in Target Names 27
4.8 Multiple Targets in a Rule 28
4.9 Multiple Rules for One Target 29

4.10 Static Pattern Rules 30
4.10.1 Syntax of Static Pattern Rules 30
4.10.2 Static Pattern Rules versus Implicit Rules ... 32
4.11 Double-Colon Rules 32
4.12 Generating Dependencies Automatically 33

5 Writing the Commands in Rules 37

5.1 Command Echoing 37
5.2 Command Execution 38
5.3 Parallel Execution 38
5.4 Errors in Commands 40
5.5 Interrupting or Killing `make` 41
5.6 Recursive Use of `make` 41
5.6.1 How the `MAKE` Variable Works 42
5.6.2 Communicating Variables to a Sub-`make` 43
5.6.3 Communicating Options to a Sub-`make` 45
5.6.4 The '`--print-directory`' Option 47
5.7 Defining Canned Command Sequences 47
5.8 Using Empty Commands 48

6 How to Use Variables 51

6.1 Basics of Variable References 51
6.2 The Two Flavors of Variables 52
6.3 Advanced Features for Reference to Variables 54
6.3.1 Substitution References 55
6.3.2 Computed Variable Names 55
6.4 How Variables Get Their Values 58
6.5 Setting Variables 58
6.6 Appending More Text to Variables 59
6.7 The `override` Directive 61
6.8 Defining Variables Verbatim 61
6.9 Variables from the Environment 62

7 Conditional Parts of Makefiles 65

7.1 Example of a Conditional 65
7.2 Syntax of Conditionals 66
7.3 Conditionals that Test Flags 69

8 Functions for Transforming Text 71

8.1 Function Call Syntax 71
8.2 Functions for String Substitution and Analysis 72
8.3 Functions for File Names 75
8.4 The `foreach` Function 78

8.5 The `origin` Function 79
8.6 The `shell` Function 80

9 How to Run `make` 83

9.1 Arguments to Specify the Makefile 83
9.2 Arguments to Specify the Goals 83
9.3 Instead of Executing the Commands 85
9.4 Avoiding Recompilation of Some Files 87
9.5 Overriding Variables 87
9.6 Testing the Compilation of a Program 88
9.7 Summary of Options 89

10 Using Implicit Rules 95

10.1 Using Implicit Rules 95
10.2 Catalogue of Implicit Rules 96
10.3 Variables Used by Implicit Rules 100
10.4 Chains of Implicit Rules 103
10.5 Defining and Redefining Pattern Rules 104
10.5.1 Introduction to Pattern Rules 104
10.5.2 Pattern Rule Examples 105
10.5.3 Automatic Variables 106
10.5.4 How Patterns Match 108
10.5.5 Match-Anything Pattern Rules 109
10.5.6 Canceling Implicit Rules 110
10.6 Defining Last-Resort Default Rules 111
10.7 Old-Fashioned Suffix Rules 111
10.8 Implicit Rule Search Algorithm 113

11 Using `make` to Update Archive Files 115

11.1 Archive Members as Targets 115
11.2 Implicit Rule for Archive Member Targets 115
11.2.1 Updating Archive Symbol Directories 116
11.3 Dangers When Using Archives 117
11.4 Suffix Rules for Archive Files 117

12 Features of GNU `make` 119

13 Incompatibilities and Missing Features ... 123

14 Makefile Conventions 125
- 14.1 General Conventions for Makefiles 125
- 14.2 Utilities in Makefiles 126
- 14.3 Standard Targets for Users 126
- 14.4 Variables for Specifying Commands 130
- 14.5 Variables for Installation Directories 131

Appendix A Quick Reference 137

Appendix B Complex Makefile Example 143

Index of Concepts 149

Index of Functions, Variables, & Directives ... 157

1 Overview of make

The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. This manual describes GNU make, which was implemented by Richard Stallman and Roland McGrath. GNU make conforms to section 6.2 of *IEEE Standard 1003.2-1992* (POSIX.2).

Our examples show C programs, since they are most common, but you can use make with any programming language whose compiler can be run with a shell command. Indeed, make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use make, you must write a file called the *makefile* that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

You can provide command line arguments to make to control which files should be recompiled, or how. See Chapter 9 “How to Run make,” page 83.

1.1 How to Read This Manual

If you are new to make, or are looking for a general introduction, read the first few sections of each chapter, skipping the later sections. In each chapter, the first few sections contain introductory or general information and the later sections contain specialized or technical information. The exception is Chapter 2 “An Introduction to Makefiles,” page 3, all of which is introductory.

If you are familiar with other make programs, see Chapter 12 “Features of GNU make,” page 119, which lists the enhancements GNU make has, and Chapter 13 “Incompatibilities and Missing Features,” page 123, which explains the few things GNU make lacks that others have.

For a quick summary, see Section 9.7 “Options Summary,” page 89, Appendix A “Quick Reference,” page 137, and Section 4.7 “Special Targets,” page 27.

1.2 Problems and Bugs

If you have problems with GNU `make` or think you’ve found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you’ve actually found a real bug. Carefully reread the documentation and see if it really says you can do what you’re trying to do. If it’s not clear whether you should be able to do something or not, report that too; it’s a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible makefile that reproduces the problem. Then send us the makefile and the exact results `make` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you’ve got a precise problem, please send electronic mail either through the Internet or via UUCP:

Internet address:
: `bug-gnu-utils@prep.ai.mit.edu`

UUCP path:
: `mit-eddie!prep.ai.mit.edu!bug-gnu-utils`

Please include the version number of `make` you are using. You can get this information with the command ‘`make --version`’. Be sure also to include the type of machine and operating system you are using. If possible, include the contents of the file ‘`config.h`’ that is generated by the configuration process.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, send a message to the bug reporting address. We cannot guarantee you’ll get help with your problem, but many seasoned `make` users read the mailing list and they will probably try to help you out. The maintainers sometimes answer such questions as well, when time permits.

2 An Introduction to Makefiles

You need a file called a *makefile* to tell `make` what to do. Most often, the makefile tells `make` how to compile and link a program.

In this chapter, we will discuss a simple makefile that describes how to compile and link a text editor which consists of eight C source files and three header files. The makefile can also tell `make` how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation). To see a more complex example of a makefile, see Appendix B “Complex Makefile,” page 143.

When `make` recompiles the editor, each changed C source file must be recompiled. If a header file has changed, each C source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

2.1 What a Rule Looks Like

A simple makefile consists of “rules” with the following shape:

```
target ... : dependencies ...
        command
        ...
        ...
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘`clean`’ (see Section 4.4 “Phony Targets,” page 24).

A *dependency* is a file that is used as input to create the target. A target often depends on several files.

A *command* is an action that `make` carries out. A rule may have more than one command, each on its own line. **Please note:** you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with dependencies and serves to create a target file if any of the dependencies change. However, the rule that specifies commands for the target need not have dependencies. For example, the rule containing the delete command associated with the target ‘`clean`’ does not have dependencies.

A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the commands on

the dependencies to create or update the target. A rule can also explain how and when to carry out an action. See Chapter 4 "Writing Rules," page 17.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

2.2 A Simple Makefile

Here is a straightforward makefile that describes the way an executable file called `edit` depends on eight object files which, in turn, depend on eight C source and three header files.

In this example, all the C files include '`defs.h`', but only those defining editing commands include '`command.h`', and only low level files that change the editor buffer include '`buffer.h`'.

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

We split each long line into two lines using backslash-newline; this is like using one long line, but is easier to read.

To use this makefile to create the executable file called '`edit`', type:

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

In the example makefile, the targets include the executable file 'edit', and the object files 'main.o' and 'kbd.o'. The dependencies are files such as 'main.c' and 'defs.h'. In fact, each '.o' file is both a target and a dependency. Commands include 'cc -c main.c' and 'cc -c kbd.c'.

When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. In addition, any dependencies that are themselves automatically generated should be updated first. In this example, 'edit' depends on each of the eight object files; the object file 'main.o' depends on the source file 'main.c' and on the header file 'defs.h'.

A shell command follows each line that contains a target and dependencies. These shell commands say how to update the target file. A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that make does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All make does is execute the commands in the rule you have specified when the target file needs to be updated.)

The target 'clean' is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, 'clean' is not a dependency of any other rule. Consequently, make never does anything with it unless you tell it specifically. Note that this rule not only is not a dependency, it also does not have any dependencies, so the only purpose of the rule is to run the specified commands. Targets that do not refer to files but are just actions are called *phony targets*. See Section 4.4 "Phony Targets," page 24, for information about this kind of target. See Section 5.4 "Errors in Commands," page 40, to see how to cause make to ignore errors from rm or any other command.

2.3 How make Processes a Makefile

By default, make starts with the first rule (not counting rules whose target names start with '.'). This is called the *default goal*. (*Goals* are the targets that make strives ultimately to update. See Section 9.2 "Arguments to Specify the Goals," page 83.)

In the simple example of the previous section, the default goal is to update the executable program 'edit'; therefore, we put that rule first.

Thus, when you give the command:

```
make
```

make reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking 'edit'; but before make can fully process this rule, it must process the rules for the files that 'edit' depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each '.o' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as dependencies, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as dependencies of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not processed, unless you tell make to do so (with a command such as make clean).

Before recompiling an object file, make considers updating its dependencies, the source file and header files. This makefile does not specify anything to be done for them—the '.c' and '.h' files are not the targets of any rules—so make does nothing for these files. But make would update automatically generated C programs, such as those made by Bison or Yacc, by their own rules at this time.

After recompiling whichever object files need it, make decides whether to relink 'edit'. This must be done if the file 'edit' does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than 'edit', so 'edit' is relinked.

Thus, if we change the file 'insert.c' and run make, make will compile that file to update 'insert.o', and then link 'edit'. If we change the file 'command.h' and run make, make will recompile the object files 'kbd.o', 'command.o' and 'files.o' and then link the file 'edit'.

2.4 Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for 'edit' (repeated here):

```
edit : main.o kbd.o command.o display.o \
              insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a variable. *Variables* allow a text string to be defined once and substituted in multiple places later (see Chapter 6 "How to Use Variables," page 51).

It is standard practice for every makefile to have a variable named `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, or `OBJ` which is a list of all object file names. We would define such a variable `objects` with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable's value by writing '`$(objects)`' (see Chapter 6 "How to Use Variables," page 51).

Here is how the complete simple makefile looks when you use a variable for the object files:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

2.5 Letting `make` Deduce the Commands

It is not necessary to spell out the commands for compiling the individual C source files, because `make` can figure them out: it has an *implicit rule* for updating a '`.o`' file from a correspondingly named '`.c`' file using a '`cc -c`' command. For example, it will use the command '`cc -c main.c -o main.o`' to compile '`main.c`' into '`main.o`'. We can therefore omit the

commands from the rules for the object files. See Chapter 10 “Using Implicit Rules,” page 95.

When a ‘.c’ file is used automatically in this way, it is also automatically added to the list of dependencies. We can therefore omit the ‘.c’ files from the dependencies, provided we omit the commands.

Here is the entire example, with both of these changes, and a variable objects as suggested above:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

This is how we would write the makefile in actual practice. (The complications associated with ‘clean’ are described elsewhere. See Section 4.4 “Phony Targets,” page 24, and Section 5.4 “Errors in Commands,” page 40.)

Because implicit rules are so convenient, they are important. You will see them used frequently.

2.6 Another Style of Makefile

When the objects of a makefile are created only by implicit rules, an alternative style of makefile is possible. In this style of makefile, you group entries by their dependencies instead of by their targets. Here is what one looks like:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here 'defs.h' is given as a dependency of all the object files; 'command.h' and 'buffer.h' are dependencies of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

2.7 Rules for Cleaning the Directory

Compiling a program is not the only thing you might want to write rules for. Makefiles commonly tell how to do a few other things besides compiling a program: for example, how to delete all the object files and executables so that the directory is 'clean'.

Here is how we could write a make rule for cleaning our example editor:

```
clean:
        rm edit $(objects)
```

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. We would do this:

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

This prevents make from getting confused by an actual file called 'clean' and causes it to continue in spite of errors from rm. (See Section 4.4 "Phony Targets," page 24, and Section 5.4 "Errors in Commands," page 40.)

A rule such as this should not be placed at the beginning of the makefile, because we do not want it to run by default! Thus, in the example makefile, we want the rule for edit, which recompiles the editor, to remain the default goal.

Since clean is not a dependency of edit, this rule will not run at all if we give the command 'make' with no arguments. In order to make

the rule run, we have to type 'make clean'. See Chapter 9 "How to Run make," page 83.

3 Writing Makefiles

The information that tells `make` how to recompile a system comes from reading a data base called the *makefile*.

3.1 What Makefiles Contain

Makefiles contain five kinds of things: *explicit rules*, *implicit rules*, *variable definitions*, *directives*, and *comments*. Rules, variables, and directives are described at length in later chapters.

- An *explicit rule* says when and how to remake one or more files, called the rule's targets. It lists the other files that the targets *depend on*, and may also give commands to use to create or update the targets. See Chapter 4 "Writing Rules," page 17.
- An *implicit rule* says when and how to remake a class of files based on their names. It describes how a target may depend on a file with a name similar to the target and gives commands to create or update such a target. See Chapter 10 "Using Implicit Rules," page 95.
- A *variable definition* is a line that specifies a text string value for a variable that can be substituted into the text later. The simple makefile example shows a variable definition for `objects` as a list of all object files (see Section 2.4 "Variables Make Makefiles Simpler," page 6).
- A *directive* is a command for `make` to do something special while reading the makefile. These include:
  - Reading another makefile (see Section 3.3 "Including Other Makefiles," page 12).
  - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see Chapter 7 "Conditional Parts of Makefiles," page 65).
  - Defining a variable from a verbatim string containing multiple lines (see Section 6.8 "Defining Variables Verbatim," page 61).
- '`#`' in a line of a makefile starts a *comment*. It and the rest of the line are ignored, except that a trailing backslash not escaped by another backslash will continue the comment across multiple lines. Comments may appear on any of the lines in the makefile, except within a `define` directive, and perhaps within commands (where the shell decides what is a comment). A line containing just a comment (with perhaps spaces before it) is effectively blank, and is ignored.

3.2 What Name to Give Your Makefile

By default, when `make` looks for the makefile, it tries the following names, in order: '`GNUmakefile`', '`makefile`' and '`Makefile`'.

Normally you should call your makefile either '`makefile`' or '`Makefile`'. (We recommend '`Makefile`' because it appears prominently near the beginning of a directory listing, right near other important files such as '`README`'.) The first name checked, '`GNUmakefile`', is not recommended for most makefiles. You should use this name if you have a makefile that is specific to GNU `make`, and will not be understood by other versions of `make`. Other `make` programs look for '`makefile`' and '`Makefile`', but not '`GNUmakefile`'.

If `make` finds none of these names, it does not use any makefile. Then you must specify a goal with a command argument, and `make` will attempt to figure out how to remake it using only its built-in implicit rules. See Chapter 10 "Using Implicit Rules," page 95.

If you want to use a nonstandard name for your makefile, you can specify the makefile name with the '`-f`' or '`--file`' option. The arguments '`-f` *name*' or '`--file=`*name*' tell `make` to read the file *name* as the makefile. If you use more than one '`-f`' or '`--file`' option, you can specify several makefiles. All the makefiles are effectively concatenated in the order specified. The default makefile names '`GNUmakefile`', '`makefile`' and '`Makefile`' are not checked automatically if you specify '`-f`' or '`--file`'.

3.3 Including Other Makefiles

The `include` directive tells `make` to suspend reading the current makefile and read one or more other makefiles before continuing. The directive is a line in the makefile that looks like this:

```
include filenames...
```

filenames can contain shell file name patterns.

Extra spaces are allowed and ignored at the beginning of the line, but a tab is not allowed. (If the line begins with a tab, it will be considered a command line.) Whitespace is required between `include` and the file names, and between file names; extra whitespace is ignored there and at the end of the directive. A comment starting with '`#`' is allowed at the end of the line. If the file names contain any variable or function references, they are expanded. See Chapter 6 "How to Use Variables," page 51.

For example, if you have three '.mk' files, 'a.mk', 'b.mk', and 'c.mk', and $(bar) expands to bish bash, then the following expression

```
include foo *.mk $(bar)
```

is equivalent to

```
include foo a.mk b.mk c.mk bish bash
```

When make processes an include directive, it suspends reading of the containing makefile and reads from each listed file in turn. When that is finished, make resumes reading the makefile in which the directive appears.

One occasion for using include directives is when several programs, handled by individual makefiles in various directories, need to use a common set of variable definitions (see Section 6.5 "Setting Variables," page 58) or pattern rules (see Section 10.5 "Defining and Redefining Pattern Rules," page 104).

Another such occasion is when you want to generate dependencies from source files automatically; the dependencies can be put in a file that is included by the main makefile. This practice is generally cleaner than that of somehow appending the dependencies to the end of the main makefile as has been traditionally done with other versions of make. See Section 4.12 "Automatic Dependencies," page 33.

If the specified name does not start with a slash, and the file is not found in the current directory, several other directories are searched. First, any directories you have specified with the '-I' or with the '--include-dir' option are searched (see Section 9.7 "Summary of Options," page 89). Then the following directories (if they exist) are searched, in this order:

'*prefix*/include' (normally '/usr/local/include')

'/usr/gnu/include',

'/usr/local/include', '/usr/include'.

If an included makefile cannot be found in any of these directories, a warning message is generated, but it is not an immediately fatal error; processing of the makefile containing the include continues. Once it has finished reading makefiles, make will try to remake any that are out of date or don't exist. See Section 3.5 "How Makefiles Are Remade," page 14. Only after it has tried to find a way to remake a makefile and failed, will make diagnose the missing makefile as a fatal error.

If you want make to simply ignore a makefile which does not exist and cannot be remade, with no error message, use the -include directive instead of include, like this:

```
-include filenames...
```

This is acts like `include` in every way except that there is no error (not even a warning) if any of the *filenames* do not exist.

3.4 The Variable MAKEFILES

If the environment variable `MAKEFILES` is defined, `make` considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive: various directories are searched for those files (see Section 3.3 "Including Other Makefiles," page 12). In addition, the default goal is never taken from one of these makefiles and it is not an error if the files listed in `MAKEFILES` are not found.

The main use of `MAKEFILES` is in communication between recursive invocations of `make` (see Section 5.6 "Recursive Use of `make`," page 41). It usually is not desirable to set the environment variable before a top-level invocation of `make`, because it is usually better not to mess with a makefile from outside. However, if you are running `make` without a specific makefile, a makefile in `MAKEFILES` can do useful things to help the built-in implicit rules work better, such as defining search paths (see Section 4.3 "Directory Search," page 20).

Some users are tempted to set `MAKEFILES` in the environment automatically on login, and program makefiles to expect this to be done. This is a very bad idea, because such makefiles will fail to work if run by anyone else. It is much better to write explicit `include` directives in the makefiles. See Section 3.3 "Including Other Makefiles," page 12.

3.5 How Makefiles Are Remade

Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want `make` to get an up-to-date version of the makefile to read in.

To this end, after reading in all makefiles, `make` will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see Chapter 10 "Using Implicit Rules," page 95), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, `make` starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)

If the makefiles specify a double-colon rule to remake a file with commands but no dependencies, that file will always be remade (see

Section 4.11 “Double-Colon,” page 32). In the case of makefiles, a makefile that has a double-colon rule with commands but no dependencies will be remade every time `make` is run, and then again after `make` starts over and reads the makefiles in again. This would cause an infinite loop: `make` would constantly remake the makefile, and never do anything else. So, to avoid this, `make` will **not** attempt to remake makefiles which are specified as double-colon targets but have no dependencies.

If you do not specify any makefiles to be read with '`-f`' or '`--file`' options, `make` will try the default makefile names; see Section 3.2 “What Name to Give Your Makefile,” page 12. Unlike makefiles explicitly requested with '`-f`' or '`--file`' options, `make` is not certain that these makefiles should exist. However, if a default makefile does not exist but can be created by running `make` rules, you probably want the rules to be run so that the makefile can be used.

Therefore, if none of the default makefiles exists, `make` will try to make each of them in the same order in which they are searched for (see Section 3.2 “What Name to Give Your Makefile,” page 12) until it succeeds in making one, or it runs out of names to try. Note that it is not an error if `make` cannot find or make any makefile; a makefile is not always necessary.

When you use the '`-t`' or '`--touch`' option (see Section 9.3 “Instead of Executing the Commands,” page 85), you would not want to use an out-of-date makefile to decide which targets to touch. So the '`-t`' option has no effect on updating makefiles; they are really updated even if '`-t`' is specified. Likewise, '`-q`' (or '`--question`') and '`-n`' (or '`--just-print`') do not prevent updating of makefiles, because an out-of-date makefile would result in the wrong output for other targets. Thus, '`make -f mfile -n foo`' will update '`mfile`', read it in, and then print the commands to update '`foo`' and its dependencies without running them. The commands printed for '`foo`' will be those specified in the updated contents of '`mfile`'.

However, on occasion you might actually wish to prevent updating of even the makefiles. You can do this by specifying the makefiles as goals in the command line as well as specifying them as makefiles. When the makefile name is specified explicitly as a goal, the options '`-t`' and so on do apply to them.

Thus, '`make -f mfile -n mfile foo`' would read the makefile '`mfile`', print the commands needed to update it without actually running them, and then print the commands needed to update '`foo`' without running them. The commands for '`foo`' will be those specified by the existing contents of '`mfile`'.

3.6 Overriding Part of Another Makefile

Sometimes it is useful to have a makefile that is mostly just like another makefile. You can often use the 'include' directive to include one in the other, and add more targets or variable definitions. However, if the two makefiles give different commands for the same target, make will not let you just do this. But there is another way.

In the containing makefile (the one that wants to include the other), you can use a match-anything pattern rule to say that to remake any target that cannot be made from the information in the containing makefile, make should look in another makefile. See Section 10.5 "Pattern Rules," page 104, for more information on pattern rules.

For example, if you have a makefile called 'Makefile' that says how to make the target 'foo' (and other targets), you can write a makefile called 'GNUmakefile' that contains:

```
foo:
        frobnicate > foo

%: force
        @$(MAKE) -f Makefile $@
force: ;
```

If you say 'make foo', make will find 'GNUmakefile', read it, and see that to make 'foo', it needs to run the command 'frobnicate > foo'. If you say 'make bar', make will find no way to make 'bar' in 'GNUmakefile', so it will use the commands from the pattern rule: 'make -f Makefile bar'. If 'Makefile' provides a rule for updating 'bar', make will apply the rule. And likewise for any other target that 'GNUmakefile' does not say how to make.

The way this works is that the pattern rule has a pattern of just '%', so it matches any target whatever. The rule specifies a dependency 'force', to guarantee that the commands will be run even if the target file already exists. We give 'force' target empty commands to prevent make from searching for an implicit rule to build it—otherwise it would apply the same match-anything rule to 'force' itself and create a dependency loop!

4 Writing Rules

A *rule* appears in the makefile and says when and how to remake certain files, called the rule's *targets* (most often only one per rule). It lists the other files that are the *dependencies* of the target, and *commands* to use to create or update the target.

The order of rules is not significant, except for determining the *default goal*: the target for `make` to consider, if you do not otherwise specify one. The default goal is the target of the first rule in the first makefile. If the first rule has multiple targets, only the first target is taken as the default. There are two exceptions: a target starting with a period is not a default unless it contains one or more slashes, '`/`', as well; and, a target that defines a pattern rule has no effect on the default goal. (See Section 10.5 "Defining and Redefining Pattern Rules," page 104.)

Therefore, we usually write the makefile so that the first rule is the one for compiling the entire program or all the programs described by the makefile (often with a target called '`all`'). See Section 9.2 "Arguments to Specify the Goals," page 83.

4.1 Rule Syntax

In general, a rule looks like this:

```
targets : dependencies
        command
        ...
```

or like this:

```
targets : dependencies ; command
        command
        ...
```

The *`targets`* are file names, separated by spaces. Wildcard characters may be used (see Section 4.2 "Using Wildcard Characters in File Names," page 18) and a name of the form '`a(m)`' represents member *m* in archive file *a* (see Section 11.1 "Archive Members as Targets," page 115). Usually there is only one target per rule, but occasionally there is a reason to have more (see Section 4.8 "Multiple Targets in a Rule," page 28).

The *`command`* lines start with a tab character. The first command may appear on the line after the dependencies, with a tab character, or may appear on the same line, with a semicolon. Either way, the effect is the same. See Chapter 5 "Writing the Commands in Rules," page 37.

Because dollar signs are used to start variable references, if you really want a dollar sign in a rule you must write two of them, '`$$`' (see Chapter 6 "How to Use Variables," page 51). You may split a long line by inserting a

backslash followed by a newline, but this is not required, as make places no limit on the length of a line in a makefile.

A rule tells make two things: when the targets are out of date, and how to update them when necessary.

The criterion for being out of date is specified in terms of the *dependencies*, which consist of file names separated by spaces. (Wildcards and archive members (see Chapter 11 “Archives,” page 115) are allowed here too.) A target is out of date if it does not exist or if it is older than any of the dependencies (by comparison of last-modification times). The idea is that the contents of the target file are computed based on information in the dependencies, so if any of the dependencies changes, the contents of the existing target file are no longer necessarily valid.

How to update is specified by *commands*. These are lines to be executed by the shell (normally ‘sh’), but with some extra features (see Chapter 5 “Writing the Commands in Rules,” page 37).

4.2 Using Wildcard Characters in File Names

A single file name can specify many files using *wildcard characters*. The wildcard characters in make are ‘*’, ‘?’ and ‘[...]’, the same as in the Bourne shell. For example, ‘*.c’ specifies a list of all the files (in the working directory) whose names end in ‘.c’.

The character ‘~’ at the beginning of a file name also has special significance. If alone, or followed by a slash, it represents your home directory. For example ‘~/bin’ expands to ‘/home/you/bin’. If the ‘~’ is followed by a word, the string represents the home directory of the user named by that word. For example ‘~john/bin’ expands to ‘/home/john/bin’.

Wildcard expansion happens automatically in targets, in dependencies, and in commands (where the shell does the expansion). In other contexts, wildcard expansion happens only if you request it explicitly with the wildcard function.

The special significance of a wildcard character can be turned off by preceding it with a backslash. Thus, ‘foo*bar’ would refer to a specific file whose name consists of ‘foo’, an asterisk, and ‘bar’.

4.2.1 Wildcard Examples

Wildcards can be used in the commands of a rule, where they are expanded by the shell. For example, here is a rule to delete all the object files:

```
clean:
        rm -f *.o
```

Wildcards are also useful in the dependencies of a rule. With the following rule in the makefile, 'make print' will print all the '.c' files that have changed since the last time you printed them:

```
print: *.c
        lpr -p $?
        touch print
```

This rule uses 'print' as an empty target file; see Section 4.6 "Empty Target Files to Record Events," page 26. (The automatic variable '$?' is used to print only those files that have changed; see Section 10.5.3 "Automatic Variables," page 106.)

Wildcard expansion does not happen when you define a variable. Thus, if you write this:

```
objects = *.o
```

then the value of the variable objects is the actual string '*.o'. However, if you use the value of objects in a target, dependency or command, wildcard expansion will take place at that time. To set objects to the expansion, instead use:

```
objects := $(wildcard *.o)
```

See Section 4.2.3 "Wildcard Function," page 20.

4.2.2 Pitfalls of Using Wildcards

Now here is an example of a naive way of using wildcard expansion, that does not do what you would intend. Suppose you would like to say that the executable file 'foo' is made from all the object files in the directory, and you write this:

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

The value of objects is the actual string '*.o'. Wildcard expansion happens in the rule for 'foo', so that each *existing* '.o' file becomes a dependency of 'foo' and will be recompiled if necessary.

But what if you delete all the '.o' files? When a wildcard matches no files, it is left as it is, so then 'foo' will depend on the oddly-named file '*.o'. Since no such file is likely to exist, make will give you an error saying it cannot figure out how to make '*.o'. This is not what you want!

Actually it is possible to obtain the desired result with wildcard expansion, but you need more sophisticated techniques, including the wildcard function and string substitution. These are described in the following section.

4.2.3 The Function `wildcard`

Wildcard expansion happens automatically in rules. But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function. If you want to do wildcard expansion in such places, you need to use the `wildcard` function, like this:

```
$(wildcard pattern...)
```

This string, used anywhere in a makefile, is replaced by a space-separated list of names of existing files that match one of the given file name patterns. If no existing file name matches a pattern, then that pattern is omitted from the output of the `wildcard` function. Note that this is different from how unmatched wildcards behave in rules, where they are used verbatim rather than ignored (see Section 4.2.2 "Wildcard Pitfall," page 19).

One use of the `wildcard` function is to get a list of all the C source files in a directory, like this:

```
$(wildcard *.c)
```

We can change the list of C source files into a list of object files by replacing the '`.o`' suffix with '`.c`' in the result, like this:

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

(Here we have used another function, `patsubst`. See Section 8.2 "Functions for String Substitution and Analysis," page 72.)

Thus, a makefile to compile all C source files in the directory and then link them together could be written as follows:

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(objects)
```

(This takes advantage of the implicit rule for compiling C programs, so there is no need to write explicit rules for compiling the files. See Section 6.2 "The Two Flavors of Variables," page 52, for an explanation of '`:=`', which is a variant of '`=`'.)

4.3 Searching Directories for Dependencies

For large systems, it is often desirable to put sources in a separate directory from the binaries. The *directory search* features of `make` facilitate this by searching several directories automatically to find a dependency. When you redistribute the files among directories, you do not need to change the individual rules, just the search paths.

4.3.1 VPATH: Search Path for All Dependencies

The value of the make variable VPATH specifies a list of directories that make should search. Most often, the directories are expected to contain dependency files that are not in the current directory; however, VPATH specifies a search list that make applies for all files, including files which are targets of rules.

Thus, if a file that is listed as a target or dependency does not exist in the current directory, make searches the directories listed in VPATH for a file with that name. If a file is found in one of them, that file becomes the dependency. Rules may then specify the names of source files in the dependencies as if they all existed in the current directory. See Section 4.3.3 “Writing Shell Commands with Directory Search,” page 23.

In the VPATH variable, directory names are separated by colons or blanks. The order in which directories are listed is the order followed by make in its search.

For example,

```
VPATH = src:../headers
```

specifies a path containing two directories, ‘src’ and ‘../headers’, which make searches in that order.

With this value of VPATH, the following rule,

```
foo.o : foo.c
```

is interpreted as if it were written like this:

```
foo.o : src/foo.c
```

assuming the file ‘foo.c’ does not exist in the current directory but is found in the directory ‘src’.

4.3.2 The vpath Directive

Similar to the VPATH variable but more selective is the vpath directive (note lower case), which allows you to specify a search path for a particular class of file names, those that match a particular pattern. Thus you can supply certain search directories for one class of file names and other directories (or none) for other file names.

There are three forms of the vpath directive:

vpath *pattern directories*
: Specify the search path *directories* for file names that match *pattern*.

  The search path, *directories*, is a list of directories to be searched, separated by colons or blanks, just like the search path used in the VPATH variable.

`vpath` *pattern*
: Clear out the search path associated with *pattern*.

`vpath`
: Clear all search paths previously specified with `vpath` directives.

A `vpath` pattern is a string containing a '`%`' character. The string must match the file name of a dependency that is being searched for, the '`%`' character matching any sequence of zero or more characters (as in pattern rules; see Section 10.5 "Defining and Redefining Pattern Rules," page 104). For example, `%.h` matches files that end in `.h`. (If there is no '`%`', the pattern must match the dependency exactly, which is not useful very often.)

'`%`' characters in a `vpath` directive's pattern can be quoted with preceding backslashes ('`\`'). Backslashes that would otherwise quote '`%`' characters can be quoted with more backslashes. Backslashes that quote '`%`' characters or other backslashes are removed from the pattern before it is compared to file names. Backslashes that are not in danger of quoting '`%`' characters go unmolested.

When a dependency fails to exist in the current directory, if the *pattern* in a `vpath` directive matches the name of the dependency file, then the *directories* in that directive are searched just like (and before) the directories in the `VPATH` variable.

For example,

```
vpath %.h ../headers
```

tells `make` to look for any dependency whose name ends in '`.h`' in the directory '`../headers`' if the file is not found in the current directory.

If several `vpath` patterns match the dependency file's name, then `make` processes each matching `vpath` directive one by one, searching all the directories mentioned in each directive. `make` handles multiple `vpath` directives in the order in which they appear in the makefile; multiple directives with the same pattern are independent of each other.

Thus,

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

will look for a file ending in '`.c`' in '`foo`', then '`blish`', then '`bar`', while

```
vpath %.c foo:bar
vpath %   blish
```

will look for a file ending in '`.c`' in '`foo`', then '`bar`', then '`blish`'.

4.3.3 Writing Shell Commands with Directory Search

When a dependency is found in another directory through directory search, this cannot change the commands of the rule; they will execute as written. Therefore, you must write the commands with care so that they will look for the dependency in the directory where `make` finds it.

This is done with the *automatic variables* such as '`$^`' (see Section 10.5.3 "Automatic Variables," page 106). For instance, the value of '`$^`' is a list of all the dependencies of the rule, including the names of the directories in which they were found, and the value of '`$@`' is the target. Thus:

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

(The variable `CFLAGS` exists so you can specify flags for C compilation by implicit rules; we use it here for consistency so it will affect all C compilations uniformly; see Section 10.3 "Variables Used by Implicit Rules," page 100.)

Often the dependencies include header files as well, which you do not want to mention in the commands. The automatic variable '`$<`' is just the first dependency:

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

4.3.4 Directory Search and Implicit Rules

The search through the directories specified in `VPATH` or with `vpath` also happens during consideration of implicit rules (see Chapter 10 "Using Implicit Rules," page 95).

For example, when a file '`foo.o`' has no explicit rule, `make` considers implicit rules, such as the built-in rule to compile '`foo.c`' if that file exists. If such a file is lacking in the current directory, the appropriate directories are searched for it. If '`foo.c`' exists (or is mentioned in the makefile) in any of the directories, the implicit rule for C compilation is applied.

The commands of implicit rules normally use automatic variables as a matter of necessity; consequently they will use the file names found by directory search with no extra effort.

4.3.5 Directory Search for Link Libraries

Directory search applies in a special way to libraries used with the linker. This special feature comes into play when you write a dependency

whose name is of the form '-l*name*'. (You can tell something strange is going on here because the dependency is normally the name of a file, and the *file name* of the library looks like 'lib*name*.a', not like '-l*name*'.)

When a dependency's name has the form '-l*name*', make handles it specially by searching for the file 'lib*name*.a' in the current directory, in directories specified by matching vpath search paths and the VPATH search path, and then in the directories '/lib', '/usr/lib', and '*prefix*/lib' (normally '/usr/local/lib').

For example,

```
foo : foo.c -lcurses
        cc $^ -o $@
```

would cause the command

```
cc foo.c /usr/lib/libcurses.a -o foo
```

to execute when 'foo' is older than 'foo.c' or '/usr/lib/libcurses.a'.

4.4 Phony Targets

A phony target is one that is not really the name of a file. It is just a name for some commands to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name, and to improve performance.

If you write a rule whose commands will not create the target file, the commands will be executed every time the target comes up for remaking. Here is an example:

```
clean:
        rm *.o temp
```

Because the rm command does not create a file named 'clean', probably no such file will ever exist. Therefore, the rm command will be executed every time you say 'make clean'.

The phony target will cease to work if anything ever does create a file named 'clean' in this directory. Since it has no dependencies, the file 'clean' would inevitably be considered up to date, and its commands would not be executed. To avoid this problem, you can explicitly declare the target to be phony, using the special target .PHONY (see Section 4.7 "Special Built-in Target Names," page 27) as follows:

```
.PHONY : clean
```

Once this is done, 'make clean' will run the commands regardless of whether there is a file named 'clean'.

Since it knows that phony targets do not name actual files that could be remade from other files, make skips the implicit rule search for phony targets (see Chapter 10 "Implicit Rules," page 95). This is why declaring

a target phony is good for performance, even if you are not worried about the actual file existing.

Thus, you first write the line that states that `clean` is a phony target, then you write the rule, like this:

```
.PHONY: clean
clean:
        rm *.o temp
```

A phony target should not be a dependency of a real target file; if it is, its commands are run every time `make` goes to update that file. As long as a phony target is never a dependency of a real target, the phony target commands will be executed only when the phony target is a specified goal (see Section 9.2 “Arguments to Specify the Goals,” page 83).

Phony targets can have dependencies. When one directory contains multiple programs, it is most convenient to describe all of the programs in one makefile ‘`./Makefile`’. Since the target remade by default will be the first one in the makefile, it is common to make this a phony target named ‘`all`’ and give it, as dependencies, all the individual programs. For example:

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

Now you can say just ‘`make`’ to remake all three programs, or specify as arguments the ones to remake (as in ‘`make prog1 prog3`’).

When one phony target is a dependency of another, it serves as a subroutine of the other. For example, here ‘`make cleanall`’ will delete the object files, the difference files, and the file ‘`program`’:

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

4.5 Rules without Commands or Dependencies

If a rule has no dependencies or commands, and the target of the rule is a nonexistent file, then make imagines this target to have been updated whenever its rule is run. This implies that all targets depending on this one will always have their commands run.

An example will illustrate this:

```
clean: FORCE
        rm $(objects)
FORCE:
```

Here the target 'FORCE' satisfies the special conditions, so the target 'clean' that depends on it is forced to run its commands. There is nothing special about the name 'FORCE', but that is one name commonly used this way.

As you can see, using 'FORCE' this way has the same results as using '.PHONY: clean'.

Using '.PHONY' is more explicit and more efficient. However, other versions of make do not support '.PHONY'; thus 'FORCE' appears in many makefiles. See Section 4.4 “Phony Targets,” page 24.

4.6 Empty Target Files to Record Events

The *empty target* is a variant of the phony target; it is used to hold commands for an action that you request explicitly from time to time. Unlike a phony target, this target file can really exist; but the file's contents do not matter, and usually are empty.

The purpose of the empty target file is to record, with its last-modification time, when the rule's commands were last executed. It does so because one of the commands is a touch command to update the target file.

The empty target file must have some dependencies. When you ask to remake the empty target, the commands are executed if any dependency is more recent than the target; in other words, if a dependency has changed since the last time you remade the target. Here is an example:

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

With this rule, 'make print' will execute the lpr command if either source file has changed since the last 'make print'. The automatic variable '$?' is used to print only those files that have changed (see Section 10.5.3 “Automatic Variables,” page 106).

4.7 Special Built-in Target Names

Certain names have special meanings if they appear as targets.

`.PHONY`
: The dependencies of the special target `.PHONY` are considered to be phony targets. When it is time to consider such a target, `make` will run its commands unconditionally, regardless of whether a file with that name exists or what its last-modification time is. See Section 4.4 “Phony Targets,” page 24.

`.SUFFIXES`
: The dependencies of the special target `.SUFFIXES` are the list of suffixes to be used in checking for suffix rules. See Section 10.7 “Old-Fashioned Suffix Rules,” page 111.

`.DEFAULT`
: The commands specified for `.DEFAULT` are used for any target for which no rules are found (either explicit rules or implicit rules). See Section 10.6 “Last Resort,” page 111. If `.DEFAULT` commands are specified, every file mentioned as a dependency, but not as a target in a rule, will have these commands executed on its behalf. See Section 10.8 “Implicit Rule Search Algorithm,” page 113.

`.PRECIOUS`
: The targets which `.PRECIOUS` depends on are given the following special treatment: if `make` is killed or interrupted during the execution of their commands, the target is not deleted. See Section 5.5 “Interrupting or Killing `make`,” page 41. Also, if the target is an intermediate file, it will not be deleted after it is no longer needed, as is normally done. See Section 10.4 “Chains of Implicit Rules,” page 103.

  You can also list the target pattern of an implicit rule (such as ‘`%.o`’) as a dependency file of the special target `.PRECIOUS` to preserve intermediate files created by rules whose target patterns match that file’s name.

`.IGNORE`
: If you specify dependencies for `.IGNORE`, then `make` will ignore errors in execution of the commands run for those particular files. The commands for `.IGNORE` are not meaningful.

  If mentioned as a target with no dependencies, `.IGNORE` says to ignore errors in execution of commands for all files. This

usage of '.IGNORE' is supported only for historical compatibility. Since this affects every command in the makefile, it is not very useful; we recommend you use the more selective ways to ignore errors in specific commands. See Section 5.4 "Errors in Commands," page 40.

`.SILENT`

If you specify dependencies for `.SILENT`, then `make` will not the print commands to remake those particular files before executing them. The commands for `.SILENT` are not meaningful.

If mentioned as a target with no dependencies, `.SILENT` says not to print any commands before executing them. This usage of '.SILENT' is supported only for historical compatibility. We recommend you use the more selective ways to silence specific commands. See Section 5.1 "Command Echoing," page 37. If you want to silence all commands for a particular run of `make`, use the '-s' or '--silent' option (see Section 9.7 "Options Summary," page 89).

`.EXPORT_ALL_VARIABLES`

Simply by being mentioned as a target, this tells `make` to export all variables to child processes by default. See Section 5.6.2 "Communicating Variables to a Sub-`make`," page 43.

Any defined implicit rule suffix also counts as a special target if it appears as a target, and so does the concatenation of two suffixes, such as '.c.o'. These targets are suffix rules, an obsolete way of defining implicit rules (but a way still widely used). In principle, any target name could be special in this way if you break it in two and add both pieces to the suffix list. In practice, suffixes normally begin with '.', so these special target names also begin with '.'. See Section 10.7 "Old-Fashioned Suffix Rules," page 111.

4.8 Multiple Targets in a Rule

A rule with multiple targets is equivalent to writing many rules, each with one target, and all identical aside from that. The same commands apply to all the targets, but their effects may vary because you can substitute the actual target name into the command using '$@'. The rule contributes the same dependencies to all the targets also.

This is useful in two cases.

- You want just dependencies, no commands. For example:

```
kbd.o command.o files.o: command.h
```

gives an additional dependency to each of the three object files mentioned.

- Similar commands work for all the targets. The commands do not need to be absolutely identical, since the automatic variable '$@' can be used to substitute the particular target to be remade into the commands (see Section 10.5.3 "Automatic Variables," page 106). For example:

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

  is equivalent to

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

  Here we assume the hypothetical program `generate` makes two types of output, one if given '-big' and one if given '-little'. See Section 8.2 "Functions for String Substitution and Analysis," page 72, for an explanation of the `subst` function.

Suppose you would like to vary the dependencies according to the target, much as the variable '$@' allows you to vary the commands. You cannot do this with multiple targets in an ordinary rule, but you can do it with a *static pattern rule*. See Section 4.10 "Static Pattern Rules," page 30.

4.9 Multiple Rules for One Target

One file can be the target of several rules. All the dependencies mentioned in all the rules are merged into one list of dependencies for the target. If the target is older than any dependency from any rule, the commands are executed.

There can only be one set of commands to be executed for a file. If more than one rule gives commands for the same file, `make` uses the last set given and prints an error message. (As a special case, if the file's name begins with a dot, no error message is printed. This odd behavior is only for compatibility with other implementations of `make`.) There is no reason to write your makefiles this way; that is why `make` gives you an error message.

An extra rule with just dependencies can be used to give a few extra dependencies to many files at once. For example, one usually has a variable named `objects` containing a list of all the compiler output files in the system being made. An easy way to say that all of them must be recompiled if 'config.h' changes is to write the following:

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

This could be inserted or taken out without changing the rules that really specify how to make the object files, making it a convenient form to use if you wish to add the additional dependency intermittently.

Another wrinkle is that the additional dependencies could be specified with a variable that you set with a command argument to make (see Section 9.5 "Overriding Variables," page 87). For example,

```
extradeps=
$(objects) : $(extradeps)
```

means that the command 'make extradeps=foo.h' will consider 'foo.h' as a dependency of each object file, but plain 'make' will not.

If none of the explicit rules for a target has commands, then make searches for an applicable implicit rule to find some commands see Chapter 10 "Using Implicit Rules," page 95).

4.10 Static Pattern Rules

Static pattern rules are rules which specify multiple targets and construct the dependency names for each target based on the target name. They are more general than ordinary rules with multiple targets because the targets do not have to have identical dependencies. Their dependencies must be *analogous*, but not necessarily *identical*.

4.10.1 Syntax of Static Pattern Rules

Here is the syntax of a static pattern rule:

```
targets ...: target-pattern: dep-patterns ...
        commands
        ...
```

The *targets* list specifies the targets that the rule applies to. The targets can contain wildcard characters, just like the targets of ordinary rules (see Section 4.2 "Using Wildcard Characters in File Names," page 18).

The *target-pattern* and *dep-patterns* say how to compute the dependencies of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *dep-patterns* to make the dependency names (one from each *dep-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the

target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target 'foo.o' matches the pattern '%.o', with 'foo' as the stem. The targets 'foo.c' and 'foo.out' do not match that pattern.

The dependency names for each target are made by substituting the stem for the '%' in each dependency pattern. For example, if one dependency pattern is '%.c', then substitution of the stem 'foo' gives the dependency name 'foo.c'. It is legitimate to write a dependency pattern that does not contain '%'; then this dependency is the same for all targets.

'%' characters in pattern rules can be quoted with preceding backslashes ('\'). Backslashes that would otherwise quote '%' characters can be quoted with more backslashes. Backslashes that quote '%' characters or other backslashes are removed from the pattern before it is compared to file names or has a stem substituted into it. Backslashes that are not in danger of quoting '%' characters go unmolested. For example, the pattern 'the\%weird\\%pattern\\' has 'the%weird\' preceding the operative '%' character, and 'pattern\\' following it. The final two backslashes are left alone because they cannot affect any '%' character.

Here is an example, which compiles each of 'foo.o' and 'bar.o' from the corresponding '.c' file:

```
objects = foo.o bar.o

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

Here '$<' is the automatic variable that holds the name of the dependency and '$@' is the automatic variable that holds the name of the target; see Section 10.5.3 "Automatic Variables," page 106.

Each target specified must match the target pattern; a warning is issued for each target that does not. If you have a list of files, only some of which will match the pattern, you can use the filter function to remove nonmatching file names (see Section 8.2 "Functions for String Substitution and Analysis," page 72):

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

In this example the result of '$(filter %.o,$(files))' is 'bar.o lose.o', and the first static pattern rule causes each of these object files to be updated by compiling the corresponding C source file. The result of '$(filter %.elc,$(files))' is 'foo.elc', so that file is made from 'foo.el'.

Another example shows how to use `$*` in static pattern rules:

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

When the `generate` command is run, `$*` will expand to the stem, either '`big`' or '`little`'.

4.10.2 Static Pattern Rules versus Implicit Rules

A static pattern rule has much in common with an implicit rule defined as a pattern rule (see Section 10.5 "Defining and Redefining Pattern Rules," page 104). Both have a pattern for the target and patterns for constructing the names of dependencies. The difference is in how `make` decides *when* the rule applies.

An implicit rule *can* apply to any target that matches its pattern, but it *does* apply only when the target has no commands otherwise specified, and only when the dependencies can be found. If more than one implicit rule appears applicable, only one applies; the choice depends on the order of rules.

By contrast, a static pattern rule applies to the precise list of targets that you specify in the rule. It cannot apply to any other target and it invariably does apply to each of the targets specified. If two conflicting rules apply, and both have commands, that's an error.

The static pattern rule can be better than an implicit rule for these reasons:

- You may wish to override the usual implicit rule for a few files whose names cannot be categorized syntactically but can be given in an explicit list.
- If you cannot be sure of the precise contents of the directories you are using, you may not be sure which other irrelevant files might lead `make` to use the wrong implicit rule. The choice might depend on the order in which the implicit rule search is done. With static pattern rules, there is no uncertainty: each rule applies to precisely the targets specified.

4.11 Double-Colon Rules

Double-colon rules are rules written with '`::`' instead of '`:`' after the target names. They are handled differently from ordinary rules when the same target appears in more than one rule.

When a target appears in multiple rules, all the rules must be the same type: all ordinary, or all double-colon. If they are double-colon, each of them is independent of the others. Each double-colon rule's commands

are executed if the target is older than any dependencies of that rule. This can result in executing none, any, or all of the double-colon rules.

Double-colon rules with the same target are in fact completely separate from one another. Each double-colon rule is processed individually, just as rules with different targets are processed.

The double-colon rules for a target are executed in the order they appear in the makefile. However, the cases where double-colon rules really make sense are those where the order of executing the commands would not matter.

Double-colon rules are somewhat obscure and not often very useful; they provide a mechanism for cases in which the method used to update a target differs depending on which dependency files caused the update, and such cases are rare.

Each double-colon rule should specify commands; if it does not, an implicit rule will be used if one applies. See Chapter 10 “Using Implicit Rules,” page 95.

4.12 Generating Dependencies Automatically

In the makefile for a program, many of the rules you need to write often say only that some object file depends on some header file. For example, if ‘main.c’ uses ‘defs.h’ via an #include, you would write:

```
main.o: defs.h
```

You need this rule so that make knows that it must remake ‘main.o’ whenever ‘defs.h’ changes. You can see that for a large program you would have to write dozens of such rules in your makefile. And, you must always be very careful to update the makefile every time you add or remove an #include.

To avoid this hassle, most modern C compilers can write these rules for you, by looking at the #include lines in the source files. Usually this is done with the ‘-M’ option to the compiler. For example, the command:

```
cc -M main.c
```

generates the output:

```
main.o : main.c defs.h
```

Thus you no longer have to write all those rules yourself. The compiler will do it for you.

Note that such a dependency constitutes mentioning ‘main.o’ in a makefile, so it can never be considered an intermediate file by implicit rule search. This means that make won’t ever remove the file after using it; see Section 10.4 “Chains of Implicit Rules,” page 103.

With old make programs, it was traditional practice to use this compiler feature to generate dependencies on demand with a command like 'make depend'. That command would create a file 'depend' containing all the automatically-generated dependencies; then the makefile could use include to read them in (see Section 3.3 "Include," page 12).

In GNU make, the feature of remaking makefiles makes this practice obsolete—you need never tell make explicitly to regenerate the dependencies, because it always regenerates any makefile that is out of date. See Section 3.5 "Remaking Makefiles," page 14.

The practice we recommend for automatic dependency generation is to have one makefile corresponding to each source file. For each source file '*name*.c' there is a makefile '*name*.d' which lists what files the object file '*name*.o' depends on. That way only the source files that have changed need to be rescanned to produce the new dependencies.

Here is the pattern rule to generate a file of dependencies (i.e., a makefile) called '*name*.d' from a C source file called '*name*.c':

```
%.d: %.c
        $(SHELL) -ec '$(CC) -M $(CPPFLAGS) $< \
            | sed '\''s/$*\\.o[ :]*/& $@/g'\'' > $@'
```

See Section 10.5 "Pattern Rules," page 104, for information on defining pattern rules. The '-e' flag to the shell makes it exit immediately if the $(CC) command fails (exits with a nonzero status). Normally the shell exits with the status of the last command in the pipeline (sed in this case), so make would not notice a nonzero status from the compiler.

With the GNU C compiler, you may wish to use the '-MM' flag instead of '-M'. This omits dependencies on system header files. See section "Options Controlling the Preprocessor" in *Using GNU CC*, for details.

The purpose of the sed command is to translate (for example):

```
main.o : main.c defs.h
```

into:

```
main.o main.d : main.c defs.h
```

This makes each '.d' file depend on all the source and header files that the corresponding '.o' file depends on. make then knows it must regenerate the dependencies whenever any of the source or header files changes.

Once you've defined the rule to remake the '.d' files, you then use the include directive to read them all in. See Section 3.3 "Include," page 12. For example:

```
sources = foo.c bar.c

include $(sources:.c=.d)
```

(This example uses a substitution variable reference to translate the list of source files 'foo.c bar.c' into a list of dependency makefiles, 'foo.d bar.d'. See Section 6.3.1 "Substitution Refs," page 55, for full information on substitution references.) Since the '.d' files are makefiles like any others, make will remake them as necessary with no further work from you. See Section 3.5 "Remaking Makefiles," page 14.

5 Writing the Commands in Rules

The commands of a rule consist of shell command lines to be executed one by one. Each command line must start with a tab, except that the first command line may be attached to the target-and-dependencies line with a semicolon in between. Blank lines and lines of just comments may appear among the command lines; they are ignored. (But beware, an apparently "blank" line that begins with a tab is *not* blank! It is an empty command; see Section 5.8 "Empty Commands," page 48.)

Users use many different shell programs, but commands in makefiles are always interpreted by '/bin/sh' unless the makefile specifies otherwise. See Section 5.2 "Command Execution," page 38.

The shell that is in use determines whether comments can be written on command lines, and what syntax they use. When the shell is '/bin/sh', a '#' starts a comment that extends to the end of the line. The '#' does not have to be at the beginning of a line. Text on a line before a '#' is not part of the comment.

5.1 Command Echoing

Normally `make` prints each command line before it is executed. We call this *echoing* because it gives the appearance that you are typing the commands yourself.

When a line starts with '@', the echoing of that line is suppressed. The '@' is discarded before the command is passed to the shell. Typically you would use this for a command whose only effect is to print something, such as an `echo` command to indicate progress through the makefile:

```
@echo About to make distribution files
```

When `make` is given the flag '-n' or '--just-print', echoing is all that happens, no execution. See Section 9.7 "Summary of Options," page 89. In this case and only this case, even the commands starting with '@' are printed. This flag is useful for finding out which commands `make` thinks are necessary without actually doing them.

The '-s' or '--silent' flag to `make` prevents all echoing, as if all commands started with '@'. A rule in the makefile for the special target `.SILENT` without dependencies has the same effect (see Section 4.7 "Special Built-in Target Names," page 27). `.SILENT` is essentially obsolete since '@' is more flexible.

5.2 Command Execution

When it is time to execute commands to update a target, they are executed by making a new subshell for each line. (In practice, `make` may take shortcuts that do not affect the results.)

Please note: this implies that shell commands such as `cd` that set variables local to each process will not affect the following command lines. If you want to use `cd` to affect the next command, put the two on a single line with a semicolon between them. Then `make` will consider them a single command and pass them, together, to a shell which will execute them in sequence. For example:

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

If you would like to split a single shell command into multiple lines of text, you must use a backslash at the end of all but the last subline. Such a sequence of lines is combined into a single line, by deleting the backslash-newline sequences, before passing it to the shell. Thus, the following is equivalent to the preceding example:

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

The program used as the shell is taken from the variable `SHELL`. By default, the program '`/bin/sh`' is used.

Unlike most variables, the variable `SHELL` is never set from the environment. This is because the `SHELL` environment variable is used to specify your personal choice of shell program for interactive use. It would be very bad for personal choices like this to affect the functioning of makefiles. See Section 6.9 "Variables from the Environment," page 62.

5.3 Parallel Execution

GNU `make` knows how to execute several commands at once. Normally, `make` will execute only one command at a time, waiting for it to finish before executing the next. However, the '`-j`' or '`--jobs`' option tells `make` to execute many commands simultaneously.

If the '`-j`' option is followed by an integer, this is the number of commands to execute at once; this is called the number of *job slots*. If there is nothing looking like an integer after the '`-j`' option, there is no limit on the number of job slots. The default number of job slots is one, which means serial execution (one thing at a time).

One unpleasant consequence of running several commands simultaneously is that output from all of the commands comes when the

commands send it, so messages from different commands may be interspersed.

Another problem is that two processes cannot both take input from the same device; so to make sure that only one command tries to take input from the terminal at once, `make` will invalidate the standard input streams of all but one running command. This means that attempting to read from standard input will usually be a fatal error (a '`Broken pipe`' signal) for most child processes if there are several.

It is unpredictable which command will have a valid standard input stream (which will come from the terminal, or wherever you redirect the standard input of `make`). The first command run will always get it first, and the first command started after that one finishes will get it next, and so on.

We will change how this aspect of `make` works if we find a better alternative. In the mean time, you should not rely on any command using standard input at all if you are using the parallel execution feature; but if you are not using this feature, then standard input works normally in all commands.

If a command fails (is killed by a signal or exits with a nonzero status), and errors are not ignored for that command (see Section 5.4 "Errors in Commands," page 40), the remaining command lines to remake the same target will not be run. If a command fails and the '`-k`' or '`--keep-going`' option was not given (see Section 9.7 "Summary of Options," page 89), `make` aborts execution. If make terminates for any reason (including a signal) with child processes running, it waits for them to finish before actually exiting.

When the system is heavily loaded, you will probably want to run fewer jobs than when it is lightly loaded. You can use the '`-l`' option to tell `make` to limit the number of jobs to run at once, based on the load average. The '`-l`' or '`--max-load`' option is followed by a floating-point number. For example,

```
-l 2.5
```

will not let `make` start more than one job if the load average is above 2.5. The '`-l`' option with no following number removes the load limit, if one was given with a previous '`-l`' option.

More precisely, when `make` goes to start up a job, and it already has at least one job running, it checks the current load average; if it is not lower than the limit given with '`-l`', `make` waits until the load average goes below that limit, or until all the other jobs finish.

By default, there is no load limit.

5.4 Errors in Commands

After each shell command returns, make looks at its exit status. If the command completed successfully, the next command line is executed in a new shell; after the last command line is finished, the rule is finished.

If there is an error (the exit status is nonzero), make gives up on the current rule, and perhaps on all rules.

Sometimes the failure of a certain command does not indicate a problem. For example, you may use the mkdir command to ensure that a directory exists. If the directory already exists, mkdir will report an error, but you probably want make to continue regardless.

To ignore errors in a command line, write a '-' at the beginning of the line's text (after the initial tab). The '-' is discarded before the command is passed to the shell for execution.

For example,

```
clean:
        -rm -f *.o
```

This causes rm to continue even if it is unable to remove a file.

When you run make with the '-i' or '--ignore-errors' flag, errors are ignored in all commands of all rules. A rule in the makefile for the special target .IGNORE has the same effect, if there are no dependencies. These ways of ignoring errors are obsolete because '-' is more flexible.

When errors are to be ignored, because of either a '-' or the '-i' flag, make treats an error return just like success, except that it prints out a message that tells you the status code the command exited with, and says that the error has been ignored.

When an error happens that make has not been told to ignore, it implies that the current target cannot be correctly remade, and neither can any other that depends on it either directly or indirectly. No further commands will be executed for these targets, since their preconditions have not been achieved.

Normally make gives up immediately in this circumstance, returning a nonzero status. However, if the '-k' or '--keep-going' flag is specified, make continues to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, 'make -k' will continue compiling other object files even though it already knows that linking them will be impossible. See Section 9.7 "Summary of Options," page 89.

The usual behavior assumes that your purpose is to get the specified targets up to date; once make learns that this is impossible, it might as well report the failure immediately. The '-k' option says that the

real purpose is to test as many of the changes made in the program as possible, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs' `compile` command passes the '`-k`' flag by default.

Usually when a command fails, if it has changed the target file at all, the file is corrupted and cannot be used—or at least it is not completely updated. Yet the file's timestamp says that it is now up to date, so the next time `make` runs, it will not try to update that file. The situation is just the same as when the command is killed by a signal; see Section 5.5 "Interrupts," page 41. So generally the right thing to do is to delete the target file if the command fails after beginning to change the file. `make` will do this if `.DELETE_ON_ERROR` appears as a target. This is almost always what you want `make` to do, but it is not historical practice; so for compatibility, you must explicitly request it.

5.5 Interrupting or Killing `make`

If `make` gets a fatal signal while a command is executing, it may delete the target file that the command was supposed to update. This is done if the target file's last-modification time has changed since `make` first checked it.

The purpose of deleting the target is to make sure that it is remade from scratch when `make` is next run. Why is this? Suppose you type `Ctrl-c` while a compiler is running, and it has begun to write an object file '`foo.o`'. The `Ctrl-c` kills the compiler, resulting in an incomplete file whose last-modification time is newer than the source file '`foo.c`'. But `make` also receives the `Ctrl-c` signal and deletes this incomplete file. If `make` did not do this, the next invocation of `make` would think that '`foo.o`' did not require updating—resulting in a strange error message from the linker when it tries to link an object file half of which is missing.

You can prevent the deletion of a target file in this way by making the special target `.PRECIOUS` depend on it. Before remaking a target, `make` checks to see whether it appears on the dependencies of `.PRECIOUS`, and thereby decides whether the target should be deleted if a signal happens. Some reasons why you might do this are that the target is updated in some atomic fashion, or exists only to record a modification-time (its contents do not matter), or must exist at all times to prevent other sorts of trouble.

5.6 Recursive Use of `make`

Recursive use of `make` means using `make` as a command in a makefile. This technique is useful when you want separate makefiles for various

subsystems that compose a larger system. For example, suppose you have a subdirectory 'subdir' which has its own makefile, and you would like the containing directory's makefile to run make on the subdirectory. You can do it by writing this:

```
subsystem:
        cd subdir; $(MAKE)
```

or, equivalently, this (see Section 9.7 "Summary of Options," page 89):

```
subsystem:
        $(MAKE) -C subdir
```

You can write recursive make commands just by copying this example, but there are many things to know about how they work and why, and about how the sub-make relates to the top-level make.

5.6.1 How the MAKE Variable Works

Recursive make commands should always use the variable MAKE, not the explicit command name 'make', as shown here:

```
subsystem:
        cd subdir; $(MAKE)
```

The value of this variable is the file name with which make was invoked. If this file name was '/bin/make', then the command executed is 'cd subdir; /bin/make'. If you use a special version of make to run the top-level makefile, the same special version will be executed for recursive invocations.

As a special feature, using the variable MAKE in the commands of a rule alters the effects of the '-t' ('--touch'), '-n' ('--just-print'), or '-q' ('--question') option. Using the MAKE variable has the same effect as using a '+' character at the beginning of the command line. See Section 9.3 "Instead of Executing the Commands," page 85.

Consider the command 'make -t' in the above example. (The '-t' option marks targets as up to date without actually running any commands; see Section 9.3 "Instead of Execution," page 85.) Following the usual definition of '-t', a 'make -t' command in the example would create a file named 'subsystem' and do nothing else. What you really want it to do is run 'cd subdir; make -t'; but that would require executing the command, and '-t' says not to execute commands.

The special feature makes this do what you want: whenever a command line of a rule contains the variable MAKE, the flags '-t', '-n' and '-q' do not apply to that line. Command lines containing MAKE are executed normally despite the presence of a flag that causes most commands not to be run. The usual MAKEFLAGS mechanism passes the flags to the sub-make (see Section 5.6.3 "Communicating Options to a Sub-make,"

page 45), so your request to touch the files, or print the commands, is propagated to the subsystem.

5.6.2 Communicating Variables to a Sub-make

Variable values of the top-level `make` can be passed to the sub-`make` through the environment by explicit request. These variables are defined in the sub-`make` as defaults, but do not override what is specified in the makefile used by the sub-`make` makefile unless you use the '`-e`' switch (see Section 9.7 "Summary of Options," page 89).

To pass down, or *export*, a variable, `make` adds the variable and its value to the environment for running each command. The sub-`make`, in turn, uses the environment to initialize its table of variable values. See Section 6.9 "Variables from the Environment," page 62.

Except by explicit request, `make` exports a variable only if it is either defined in the environment initially or set on the command line, and if its name consists only of letters, numbers, and underscores. Some shells cannot cope with environment variable names consisting of characters other than letters, numbers, and underscores.

The special variables `SHELL` and `MAKEFLAGS` are always exported (unless you unexport them). `MAKEFILES` is exported if you set it to anything.

`make` automatically passes down variable values that were defined on the command line, by putting them in the `MAKEFLAGS` variable. See the next section.

Variables are *not* normally passed down if they were created by default by `make` (see Section 10.3 "Variables Used by Implicit Rules," page 100). The sub-`make` will define these for itself.

If you want to export specific variables to a sub-`make`, use the `export` directive, like this:

```
export variable ...
```

If you want to *prevent* a variable from being exported, use the `unexport` directive, like this:

```
unexport variable ...
```

As a convenience, you can define a variable and export it at the same time by doing:

```
export variable = value
```

has the same result as:

```
variable = value
export variable
```

and

```
export variable := value
```

has the same result as:

```
variable := value
export variable
```

Likewise,

```
export variable += value
```

is just like:

```
variable += value
export variable
```

See Section 6.6 “Appending More Text to Variables,” page 59.

You may notice that the `export` and `unexport` directives work in `make` in the same way they work in the shell, `sh`.

If you want all variables to be exported by default, you can use `export` by itself:

```
export
```

This tells `make` that variables which are not explicitly mentioned in an `export` or `unexport` directive should be exported. Any variable given in an `unexport` directive will still *not* be exported. If you use `export` by itself to export variables by default, variables whose names contain characters other than alphanumerics and underscores will not be exported unless specifically mentioned in an `export` directive.

The behavior elicited by an `export` directive by itself was the default in older versions of GNU `make`. If your makefiles depend on this behavior and you want to be compatible with old versions of `make`, you can write a rule for the special target `.EXPORT_ALL_VARIABLES` instead of using the `export` directive. This will be ignored by old `make`s, while the `export` directive will cause a syntax error.

Likewise, you can use `unexport` by itself to tell `make` *not* to export variables by default. Since this is the default behavior, you would only need to do this if `export` had been used by itself earlier (in an included makefile, perhaps). You **cannot** use `export` and `unexport` by themselves to have variables exported for some commands and not for others. The last `export` or `unexport` directive that appears by itself determines the behavior for the entire run of `make`.

As a special feature, the variable `MAKELEVEL` is changed when it is passed down from level to level. This variable's value is a string which is the depth of the level as a decimal number. The value is ‘`0`’ for the top-level `make`; ‘`1`’ for a sub-`make`, ‘`2`’ for a sub-sub-`make`, and so on. The incrementation happens when `make` sets up the environment for a command.

The main use of `MAKELEVEL` is to test it in a conditional directive (see Chapter 7 “Conditional Parts of Makefiles,” page 65); this way you can

write a makefile that behaves one way if run recursively and another way if run directly by you.

You can use the variable `MAKEFILES` to cause all sub-`make` commands to use additional makefiles. The value of `MAKEFILES` is a whitespace-separated list of file names. This variable, if defined in the outer-level makefile, is passed down through the environment; then it serves as a list of extra makefiles for the sub-`make` to read before the usual or specified ones. See Section 3.4 “The Variable `MAKEFILES`,” page 14.

5.6.3 Communicating Options to a Sub-`make`

Flags such as '`-s`' and '`-k`' are passed automatically to the sub-`make` through the variable `MAKEFLAGS`. This variable is set up automatically by `make` to contain the flag letters that `make` received. Thus, if you do '`make -ks`' then `MAKEFLAGS` gets the value '`ks`'.

As a consequence, every sub-`make` gets a value for `MAKEFLAGS` in its environment. In response, it takes the flags from that value and processes them as if they had been given as arguments. See Section 9.7 “Summary of Options,” page 89.

Likewise variables defined on the command line are passed to the sub-`make` through `MAKEFLAGS`. Words in the value of `MAKEFLAGS` that contain '`=`', `make` treats as variable definitions just as if they appeared on the command line. See Section 9.5 “Overriding Variables,” page 87.

The options '`-C`', '`-f`', '`-o`', and '`-W`' are not put into `MAKEFLAGS`; these options are not passed down.

The '`-j`' option is a special case (see Section 5.3 “Parallel Execution,” page 38). If you set it to some numeric value, '`-j 1`' is always put into `MAKEFLAGS` instead of the value you specified. This is because if the '`-j`' option were passed down to sub-`make`s, you would get many more jobs running in parallel than you asked for. If you give '`-j`' with no numeric argument, meaning to run as many jobs as possible in parallel, this is passed down, since multiple infinities are no more than one.

If you do not want to pass the other flags down, you must change the value of `MAKEFLAGS`, like this:

```
MAKEFLAGS=
subsystem:
        cd subdir; $(MAKE)
```

or like this:

```
subsystem:
        cd subdir; $(MAKE) MAKEFLAGS=
```

The command line variable definitions really appear in the variable `MAKEOVERRIDES`, and `MAKEFLAGS` contains a reference to this variable. If you do want to pass flags down normally, but don't want to pass down

the command line variable definitions, you can reset MAKEOVERRIDES to empty, like this:

```
MAKEOVERRIDES =
```

This is not usually useful to do. However, some systems have a small fixed limit on the size of the environment, and putting so much information in into the value of MAKEFLAGS can exceed it. If you see the error message 'Arg list too long', this may be the problem.
(For strict compliance with POSIX.2, changing MAKEOVERRIDES does not affect MAKEFLAGS if the special target '.POSIX' appears in the makefile. You probably do not care about this.)

A similar variable MFLAGS exists also, for historical compatibility. It has the same value as MAKEFLAGS except that it does not contain the command line variable definitions, and it always begins with a hyphen unless it is empty (MAKEFLAGS begins with a hyphen only when it begins with an option that has no single-letter version, such as '--warn-undefined-variables'). MFLAGS was traditionally used explicitly in the recursive make command, like this:

```
subsystem:
        cd subdir; $(MAKE) $(MFLAGS)
```

but now MAKEFLAGS makes this usage redundant. If you want your makefiles to be compatible with old make programs, use this technique; it will work fine with more modern make versions too.

The MAKEFLAGS variable can also be useful if you want to have certain options, such as '-k' (see Section 9.7 "Summary of Options," page 89), set each time you run make. You simply put a value for MAKEFLAGS in your environment. You can also set MAKEFLAGS in a makefile, to specify additional flags that should also be in effect for that makefile. (Note that you cannot use MFLAGS this way. That variable is set only for compatibility; make does not interpret a value you set for it in any way.)

When make interprets the value of MAKEFLAGS (either from the environment or from a makefile), it first prepends a hyphen if the value does not already begin with one. Then it chops the value into words separated by blanks, and parses these words as if they were options given on the command line (except that '-C', '-f', '-h', '-o', '-W', and their long-named versions are ignored; and there is no error for an invalid option).

If you do put MAKEFLAGS in your environment, you should be sure not to include any options that will drastically affect the actions of make and undermine the purpose of makefiles and of make itself. For instance, the '-t', '-n', and '-q' options, if put in one of these variables, could have disastrous consequences and would certainly have at least surprising and probably annoying effects.

5.6.4 The '--print-directory' Option

If you use several levels of recursive make invocations, the '-w' or '--print-directory' option can make the output a lot easier to understand by showing each directory as make starts processing it and as make finishes processing it. For example, if 'make -w' is run in the directory '/u/gnu/make', make will print a line of the form:

```
make: Entering directory `/u/gnu/make'.
```

before doing anything else, and a line of the form:

```
make: Leaving directory `/u/gnu/make'.
```

when processing is completed.

Normally, you do not need to specify this option because 'make' does it for you: '-w' is turned on automatically when you use the '-C' option, and in sub-makes. make will not automatically turn on '-w' if you also use '-s', which says to be silent, or if you use '--no-print-directory' to explicitly disable it.

5.7 Defining Canned Command Sequences

When the same sequence of commands is useful in making various targets, you can define it as a canned sequence with the define directive, and refer to the canned sequence from the rules for those targets. The canned sequence is actually a variable, so the name must not conflict with other variable names.

Here is an example of defining a canned sequence of commands:

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

Here run-yacc is the name of the variable being defined; endef marks the end of the definition; the lines in between are the commands. The define directive does not expand variable references and function calls in the canned sequence; the '$' characters, parentheses, variable names, and so on, all become part of the value of the variable you are defining. See Section 6.8 "Defining Variables Verbatim," page 61, for a complete explanation of define.

The first command in this example runs Yacc on the first dependency of whichever rule uses the canned sequence. The output file from Yacc is always named 'y.tab.c'. The second command moves the output to the rule's target file name.

To use the canned sequence, substitute the variable into the commands of a rule. You can substitute it like any other variable (see Section 6.1 "Basics of Variable References," page 51). Because variables

defined by `define` are recursively expanded variables, all the variable references you wrote inside the `define` are expanded now. For example:

```
foo.c : foo.y
        $(run-yacc)
```

'`foo.y`' will be substituted for the variable '`$^`' when it occurs in `run-yacc`'s value, and '`foo.c`' for '`$@`'.

This is a realistic example, but this particular one is not needed in practice because `make` has an implicit rule to figure out these commands based on the file names involved (see Chapter 10 "Using Implicit Rules," page 95).

In command execution, each line of a canned sequence is treated just as if the line appeared on its own in the rule, preceded by a tab. In particular, `make` invokes a separate subshell for each line. You can use the special prefix characters that affect command lines ('`@`', '`-`', and '`+`') on each line of a canned sequence. See Chapter 5 "Writing the Commands in Rules," page 37. For example, using this canned sequence:

```
define frobnicate
@echo "frobnicating target $@"
frob-step-1 $< -o $@-step-1
frob-step-2 $@-step-1 -o $@
endef
```

`make` will not echo the first line, the `echo` command. But it *will* echo the following two command lines.

On the other hand, prefix characters on the command line that refers to a canned sequence apply to every line in the sequence. So the rule:

```
frob.out: frob.in
@$(frobnicate)
```

does not echo *any* commands. (See Section 5.1 "Command Echoing," page 37, for a full explanation of '`@`'.)

5.8 Using Empty Commands

It is sometimes useful to define commands which do nothing. This is done simply by giving a command that consists of nothing but whitespace. For example:

```
target: ;
```

defines an empty command string for '`target`'. You could also use a line beginning with a tab character to define an empty command string, but this would be confusing because such a line looks empty.

You may be wondering why you would want to define a command string that does nothing. The only reason this is useful is to prevent a target from getting implicit commands (from implicit rules or the

`.DEFAULT` special target; see Chapter 10 “Implicit Rules,” page 95 and see Section 10.6 “Defining Last-Resort Default Rules,” page 111).

You may be inclined to define empty command strings for targets that are not actual files, but only exist so that their dependencies can be remade. However, this is not the best way to do that, because the dependencies may not be remade properly if the target file actually does exist. See Section 4.4 “Phony Targets,” page 24, for a better way to do this.

6 How to Use Variables

A *variable* is a name defined in a makefile to represent a string of text, called the variable's *value*. These values are substituted by explicit request into targets, dependencies, commands, and other parts of the makefile. (In some other versions of `make`, variables are called *macros*.)

Variables and functions in all parts of a makefile are expanded when read, except for the shell commands in rules, the right-hand sides of variable definitions using '`=`', and the bodies of variable definitions using the `define` directive.

Variables can represent lists of file names, options to pass to compilers, programs to run, directories to look in for source files, directories to write output in, or anything else you can imagine.

A variable name may be any sequence of characters not containing '`:`', '`#`', '`=`', or leading or trailing whitespace. However, variable names containing characters other than letters, numbers, and underscores should be avoided, as they may be given special meanings in the future, and with some shells they cannot be passed through the environment to a sub-`make` (see Section 5.6.2 "Communicating Variables to a Sub-`make`," page 43).

Variable names are case-sensitive. The names '`foo`', '`FOO`', and '`Foo`' all refer to different variables.

It is traditional to use upper case letters in variable names, but we recommend using lower case letters for variable names that serve internal purposes in the makefile, and reserving upper case for parameters that control implicit rules or for parameters that the user should override with command options (see Section 9.5 "Overriding Variables," page 87).

A few variables have names that are a single punctuation character or just a few characters. These are the *automatic variables*, and they have particular specialized uses. See Section 10.5.3 "Automatic Variables," page 106.

6.1 Basics of Variable References

To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either '`$(foo)`' or '`${foo}`' is a valid reference to the variable `foo`. This special significance of '`$`' is why you must write '`$$`' to have the effect of a single dollar sign in a file name or command.

Variable references can be used in any context: targets, dependencies, commands, most directives, and new variable values. Here is an example

of a common case, where a variable holds the names of all the object files in a program:

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

Variable references work by strict textual substitution. Thus, the rule

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

could be used to compile a C program 'prog.c'. Since spaces before the variable value are ignored in variable assignments, the value of foo is precisely 'c'. (Don't actually write your makefiles this way!)

A dollar sign followed by a character other than a dollar sign, open-parenthesis or open-brace treats that single character as the variable name. Thus, you could reference the variable x with '$x'. However, this practice is strongly discouraged, except in the case of the automatic variables (see Section 10.5.3 "Automatic Variables," page 106).

6.2 The Two Flavors of Variables

There are two ways that a variable in GNU make can have a value; we call them the two *flavors* of variables. The two flavors are distinguished in how they are defined and in what they do when expanded.

The first flavor of variable is a *recursively expanded* variable. Variables of this sort are defined by lines using '=' (see Section 6.5 "Setting Variables," page 58) or by the define directive (see Section 6.8 "Defining Variables Verbatim," page 61). The value you specify is installed verbatim; if it contains references to other variables, these references are expanded whenever this variable is substituted (in the course of expanding some other string). When this happens, it is called *recursive expansion*.

For example,

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

will echo 'Huh?': '$(foo)' expands to '$(bar)' which expands to '$(ugh)' which finally expands to 'Huh?'.

This flavor of variable is the only sort supported by other versions of make. It has its advantages and its disadvantages. An advantage (most would say) is that:

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

will do what was intended: when 'CFLAGS' is expanded in a command, it will expand to '-Ifoo -Ibar -O'. A major disadvantage is that you cannot append something on the end of a variable, as in

```
CFLAGS = $(CFLAGS) -O
```

because it will cause an infinite loop in the variable expansion. (Actually make detects the infinite loop and reports an error.)

Another disadvantage is that any functions (see Chapter 8 "Functions for Transforming Text," page 71) referenced in the definition will be executed every time the variable is expanded. This makes make run slower; worse, it causes the wildcard and shell functions to give unpredictable results because you cannot easily control when they are called, or even how many times.

To avoid all the problems and inconveniences of recursively expanded variables, there is another flavor: simply expanded variables.

Simply expanded variables are defined by lines using ':=' (see Section 6.5 "Setting Variables," page 58). The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions, when the variable is defined. The actual value of the simply expanded variable is the result of expanding the text that you write. It does not contain any references to other variables; it contains their values *as of the time this variable was defined*. Therefore,

```
x := foo
y := $(x) bar
x := later
```

is equivalent to

```
y := foo bar
x := later
```

When a simply expanded variable is referenced, its value is substituted verbatim.

Here is a somewhat more complicated example, illustrating the use of ':=' in conjunction with the shell function. (See Section 8.6 "The shell Function," page 80.) This example also shows use of the variable MAKELEVEL, which is changed when it is passed down from level to level. (See Section 5.6.2 "Communicating Variables to a Sub-make," page 43, for information about MAKELEVEL.)

```
ifeq (0,${MAKELEVEL})
cur-dir   := $(shell pwd)
whoami    := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

An advantage of this use of ':=' is that a typical 'descend into a directory' command then looks like this:

```
${subdirs}:
      ${MAKE} cur-dir=${cur-dir}/$@ -C $@ all
```

Simply expanded variables generally make complicated makefile programming more predictable because they work like variables in most programming languages. They allow you to redefine a variable using its own value (or its value processed in some way by one of the expansion functions) and to use the expansion functions much more efficiently (see Chapter 8 "Functions for Transforming Text," page 71).

You can also use them to introduce controlled leading whitespace into variable values. Leading whitespace characters are discarded from your input before substitution of variable references and function calls; this means you can include leading spaces in a variable value by protecting them with variable references, like this:

```
nullstring :=
space := $(nullstring) # end of the line
```

Here the value of the variable `space` is precisely one space. The comment '`# end of the line`' is included here just for clarity. Since trailing space characters are *not* stripped from variable values, just a space at the end of the line would have the same effect (but be rather hard to read). If you put whitespace at the end of a variable value, it is a good idea to put a comment like that at the end of the line to make your intent clear. Conversely, if you do *not* want any whitespace characters at the end of your variable value, you must remember not to put a random comment on the end of the line after some whitespace, such as this:

```
dir := /foo/bar    # directory to put the frobs in
```

Here the value of the variable `dir` is '`/foo/bar    `' (with four trailing spaces), which was probably not the intention. (Imagine something like '`$(dir)/file`' with this definition!)

6.3 Advanced Features for Reference to Variables

This section describes some advanced features you can use to reference variables in more flexible ways.

6.3.1 Substitution References

A *substitution reference* substitutes the value of a variable with alterations that you specify. It has the form '`$(`*var*`:`*a*`=`*b*`)`' (or '`${`*var*`:`*a*`=`*b*`}`') and its meaning is to take the value of the variable *var*, replace every *a* at the end of a word with *b* in that value, and substitute the resulting string.

When we say "at the end of a word", we mean that *a* must appear either followed by whitespace or at the end of the value in order to be replaced; other occurrences of *a* in the value are unaltered. For example:

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

sets '`bar`' to '`a.c b.c c.c`'. See Section 6.5 "Setting Variables," page 58.

A substitution reference is actually an abbreviation for use of the `patsubst` expansion function (see Section 8.2 "Functions for String Substitution and Analysis," page 72). We provide substitution references as well as `patsubst` for compatibility with other implementations of `make`.

Another type of substitution reference lets you use the full power of the `patsubst` function. It has the same form '`$(`*var*`:`*a*`=`*b*`)`' described above, except that now *a* must contain a single '`%`' character. This case is equivalent to '`$(patsubst` *a*`,`*b*`,$(`*var*`))`'. See Section 8.2 "Functions for String Substitution and Analysis," page 72, for a description of the `patsubst` function.

For example:

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

sets '`bar`' to '`a.c b.c c.c`'.

6.3.2 Computed Variable Names

Computed variable names are a complicated concept needed only for sophisticated makefile programming. For most purposes you need not consider them, except to know that making a variable with a dollar sign in its name might have strange results. However, if you are the type that wants to understand everything, or you are actually interested in what they do, read on.

Variables may be referenced inside the name of a variable. This is called a *computed variable name* or a *nested variable reference*. For example,

```
x = y
y = z
a := $($(x))
```

defines a as ‘z’: the ‘$(x)’ inside ‘$($(x))’ expands to ‘y’, so ‘$($(x))’ expands to ‘$(y)’ which in turn expands to ‘z’. Here the name of the variable to reference is not stated explicitly; it is computed by expansion of ‘$(x)’. The reference ‘$(x)’ here is nested within the outer variable reference.

The previous example shows two levels of nesting, but any number of levels is possible. For example, here are three levels:

```
x = y
y = z
z = u
a := $($($(x)))
```

Here the innermost ‘$(x)’ expands to ‘y’, so ‘$($(x))’ expands to ‘$(y)’ which in turn expands to ‘z’; now we have ‘$(z)’, which becomes ‘u’.

References to recursively-expanded variables within a variable name are reexpanded in the usual fashion. For example:

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

defines a as ‘Hello’: ‘$($(x))’ becomes ‘$($(y))’ which becomes ‘$(z)’ which becomes ‘Hello’.

Nested variable references can also contain modified references and function invocations (see Chapter 8 “Functions for Transforming Text,” page 71), just like any other reference. For example, using the subst function (see Section 8.2 “Functions for String Substitution and Analysis,” page 72):

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

eventually defines a as ‘Hello’. It is doubtful that anyone would ever want to write a nested reference as convoluted as this one, but it works: ‘$($($(z)))’ expands to ‘$($(y))’ which becomes ‘$($(subst 1,2,$(x)))’. This gets the value ‘variable1’ from x and changes it by substitution to ‘variable2’, so that the entire string becomes ‘$(variable2)’, a simple variable reference whose value is ‘Hello’.

A computed variable name need not consist entirely of a single variable reference. It can contain several variable references, as well as some invariant text. For example,

```
a_dirs := dira dirb
1_dirs := dir1 dir2
```

```
a_files := filea fileb
1_files := file1 file2

ifeq "$(use_a)" "yes"
a1 := a
else
a1 := 1
endif

ifeq "$(use_dirs)" "yes"
df := dirs
else
df := files
endif

dirs := $($(a1)_$(df))
```

will give `dirs` the same value as `a_dirs`, `1_dirs`, `a_files` or `1_files` depending on the settings of `use_a` and `use_dirs`.

Computed variable names can also be used in substitution references:

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_objects:.o=.c)
```

defines `sources` as either '`a.c b.c c.c`' or '`1.c 2.c 3.c`', depending on the value of `a1`.

The only restriction on this sort of use of nested variable references is that they cannot specify part of the name of a function to be called. This is because the test for a recognized function name is done before the expansion of nested references. For example,

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

attempts to give '`foo`' the value of the variable '`sort a d b g q c`' or '`strip a d b g q c`', rather than giving '`a d b g q c`' as the argument to either the `sort` or the `strip` function. This restriction could be removed in the future if that change is shown to be a good idea.

You can also use computed variable names in the left-hand side of a variable assignment, or in a `define` directive, as in:

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```

This example defines the variables ‘dir’, ‘foo_sources’, and ‘foo_print’.

Note that *nested variable references* are quite different from *recursively expanded variables* (see Section 6.2 “The Two Flavors of Variables,” page 52), though both are used together in complex ways when doing makefile programming.

6.4 How Variables Get Their Values

Variables can get values in several different ways:

- You can specify an overriding value when you run make. See Section 9.5 “Overriding Variables,” page 87.
- You can specify a value in the makefile, either with an assignment (see Section 6.5 “Setting Variables,” page 58) or with a verbatim definition (see Section 6.8 “Defining Variables Verbatim,” page 61).
- Variables in the environment become make variables. See Section 6.9 “Variables from the Environment,” page 62.
- Several *automatic* variables are given new values for each rule. Each of these has a single conventional use. See Section 10.5.3 “Automatic Variables,” page 106.
- Several variables have constant initial values. See Section 10.3 “Variables Used by Implicit Rules,” page 100.

6.5 Setting Variables

To set a variable from the makefile, write a line starting with the variable name followed by ‘=’ or ‘:=’. Whatever follows the ‘=’ or ‘:=’ on the line becomes the value. For example,

```
objects = main.o foo.o bar.o utils.o
```

defines a variable named objects. Whitespace around the variable name and immediately after the ‘=’ is ignored.

Variables defined with ‘=’ are *recursively expanded* variables. Variables defined with ‘:=’ are *simply expanded* variables; these definitions can contain variable references which will be expanded before the definition is made. See Section 6.2 “The Two Flavors of Variables,” page 52.

The variable name may contain function and variable references, which are expanded when the line is read to find the actual variable name to use.

There is no limit on the length of the value of a variable except the amount of swapping space on the computer. When a variable definition is long, it is a good idea to break it into several lines by inserting backslash-newline at convenient places in the definition. This will not affect the functioning of `make`, but it will make the makefile easier to read.

Most variable names are considered to have the empty string as a value if you have never set them. Several variables have built-in initial values that are not empty, but you can set them in the usual ways (see Section 10.3 "Variables Used by Implicit Rules," page 100). Several special variables are set automatically to a new value for each rule; these are called the *automatic* variables (see Section 10.5.3 "Automatic Variables," page 106).

6.6 Appending More Text to Variables

Often it is useful to add more text to the value of a variable already defined. You do this with a line containing '+=', like this:

```
objects += another.o
```

This takes the value of the variable `objects`, and adds the text '`another.o`' to it (preceded by a single space). Thus:

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

sets `objects` to '`main.o foo.o bar.o utils.o another.o`'.

Using '+=' is similar to:

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

but differs in ways that become important when you use more complex values.

When the variable in question has not been defined before, '+=' acts just like normal '=': it defines a recursively-expanded variable. However, when there *is* a previous definition, exactly what '+=' does depends on what flavor of variable you defined originally. See Section 6.2 "The Two Flavors of Variables," page 52, for an explanation of the two flavors of variables.

When you add to a variable's value with '+=', `make` acts essentially as if you had included the extra text in the initial definition of the variable. If you defined it first with ':=', making it a simply-expanded variable, '+=' adds to that simply-expanded definition, and expands the new text

before appending it to the old value just as ':=' does (see Section 6.5 "Setting Variables," page 58, for a full explanation of ':='). In fact,

```
variable := value
variable += more
```

is exactly equivalent to:

```
variable := value
variable := $(variable) more
```

On the other hand, when you use '+=' with a variable that you defined first to be recursively-expanded using plain '=', make does something a bit different. Recall that when you define a recursively-expanded variable, make does not expand the value you set for variable and function references immediately. Instead it stores the text verbatim, and saves these variable and function references to be expanded later, when you refer to the new variable (see Section 6.2 "The Two Flavors of Variables," page 52). When you use '+=' on a recursively-expanded variable, it is this unexpanded text to which make appends the new text you specify.

```
variable = value
variable += more
```

is roughly equivalent to:

```
temp = value
variable = $(temp) more
```

except that of course it never defines a variable called temp. The importance of this comes when the variable's old value contains variable references. Take this common example:

```
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
```

The first line defines the CFLAGS variable with a reference to another variable, includes. (CFLAGS is used by the rules for C compilation; see Section 10.2 "Catalogue of Implicit Rules," page 96.) Using '=' for the definition makes CFLAGS a recursively-expanded variable, meaning '$(includes) -O' is *not* expanded when make processes the definition of CFLAGS. Thus, includes need not be defined yet for its value to take effect. It only has to be defined before any reference to CFLAGS. If we tried to append to the value of CFLAGS without using '+=', we might do it like this:

```
CFLAGS := $(CFLAGS) -pg # enable profiling
```

This is pretty close, but not quite what we want. Using ':=' redefines CFLAGS as a simply-expanded variable; this means make expands the text '$(CFLAGS) -pg' before setting the variable. If includes is not yet defined, we get ' -O -pg', and a later definition of includes will have no effect. Conversely, by using '+=' we set CFLAGS to the *unexpanded* value '$(includes) -O -pg'. Thus we preserve the reference to includes, so if

that variable gets defined at any later point, a reference like '$(CFLAGS)' still uses its value.

6.7 The override Directive

If a variable has been set with a command argument (see Section 9.5 "Overriding Variables," page 87), then ordinary assignments in the makefile are ignored. If you want to set the variable in the makefile even though it was set with a command argument, you can use an override directive, which is a line that looks like this:

```
override variable = value
```

or

```
override variable := value
```

To append more text to a variable defined on the command line, use:

```
override variable += more text
```

See Section 6.6 "Appending More Text to Variables," page 59.

The override directive was not invented for escalation in the war between makefiles and command arguments. It was invented so you can alter and add to values that the user specifies with command arguments.

For example, suppose you always want the '-g' switch when you run the C compiler, but you would like to allow the user to specify the other switches with a command argument just as usual. You could use this override directive:

```
override CFLAGS += -g
```

You can also use override directives with define directives. This is done as you might expect:

```
override define foo
bar
endef
```

See the next section for information about define.

6.8 Defining Variables Verbatim

Another way to set the value of a variable is to use the define directive. This directive has an unusual syntax which allows newline characters to be included in the value, which is convenient for defining canned sequences of commands (see Section 5.7 "Defining Canned Command Sequences," page 47).

The define directive is followed on the same line by the name of the variable and nothing more. The value to give the variable appears on the following lines. The end of the value is marked by a line containing

just the word `endef`. Aside from this difference in syntax, `define` works just like '=': it creates a recursively-expanded variable (see Section 6.2 "The Two Flavors of Variables," page 52). The variable name may contain function and variable references, which are expanded when the directive is read to find the actual variable name to use.

```
define two-lines
echo foo
echo $(bar)
endef
```

The value in an ordinary assignment cannot contain a newline; but the newlines that separate the lines of the value in a `define` become part of the variable's value (except for the final newline which precedes the `endef` and is not considered part of the value).

The previous example is functionally equivalent to this:

```
two-lines = echo foo; echo $(bar)
```

since two commands separated by semicolon behave much like two separate shell commands. However, note that using two separate lines means `make` will invoke the shell twice, running an independent subshell for each line. See Section 5.2 "Command Execution," page 38.

If you want variable definitions made with `define` to take precedence over command-line variable definitions, you can use the `override` directive together with `define`:

```
override define two-lines
foo
$(bar)
endef
```

See Section 6.7 "The `override` Directive," page 61.

6.9 Variables from the Environment

Variables in `make` can come from the environment in which `make` is run. Every environment variable that `make` sees when it starts up is transformed into a `make` variable with the same name and value. But an explicit assignment in the makefile, or with a command argument, overrides the environment. (If the '-e' flag is specified, then values from the environment override assignments in the makefile. See Section 9.7 "Summary of Options," page 89. But this is not recommended practice.)

Thus, by setting the variable `CFLAGS` in your environment, you can cause all C compilations in most makefiles to use the compiler switches you prefer. This is safe for variables with standard or conventional meanings because you know that no makefile will use them for other things. (But this is not totally reliable; some makefiles set `CFLAGS` explicitly and therefore are not affected by the value in the environment.)

When `make` is invoked recursively, variables defined in the outer invocation can be passed to inner invocations through the environment (see Section 5.6 “Recursive Use of `make`,” page 41). By default, only variables that came from the environment or the command line are passed to recursive invocations. You can use the `export` directive to pass other variables. See Section 5.6.2 “Communicating Variables to a Sub-`make`,” page 43, for full details.

Other use of variables from the environment is not recommended. It is not wise for makefiles to depend for their functioning on environment variables set up outside their control, since this would cause different users to get different results from the same makefile. This is against the whole purpose of most makefiles.

Such problems would be especially likely with the variable `SHELL`, which is normally present in the environment to specify the user's choice of interactive shell. It would be very undesirable for this choice to affect `make`. So `make` ignores the environment value of `SHELL`.

7 Conditional Parts of Makefiles

A *conditional* causes part of a makefile to be obeyed or ignored depending on the values of variables. Conditionals can compare the value of one variable to another, or the value of a variable to a constant string. Conditionals control what make actually "sees" in the makefile, so they *cannot* be used to control shell commands at the time of execution.

7.1 Example of a Conditional

The following example of a conditional tells make to use one set of libraries if the CC variable is 'gcc', and a different set of libraries otherwise. It works by controlling which of two command lines will be used as the command for a rule. The result is that 'CC=gcc' as an argument to make changes not only which compiler is used but also which libraries are linked.

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

This conditional uses three directives: one ifeq, one else and one endif.

The ifeq directive begins the conditional, and specifies the condition. It contains two arguments, separated by a comma and surrounded by parentheses. Variable substitution is performed on both arguments and then they are compared. The lines of the makefile following the ifeq are obeyed if the two arguments match; otherwise they are ignored.

The else directive causes the following lines to be obeyed if the previous conditional failed. In the example above, this means that the second alternative linking command is used whenever the first alternative is not used. It is optional to have an else in a conditional.

The endif directive ends the conditional. Every conditional must end with an endif. Unconditional makefile text follows.

As this example illustrates, conditionals work at the textual level: the lines of the conditional are treated as part of the makefile, or ignored, according to the condition. This is why the larger syntactic units of the makefile, such as rules, may cross the beginning or the end of the conditional.

When the variable CC has the value 'gcc', the above example has this effect:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

When the variable CC has any other value, the effect is this:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

Equivalent results can be obtained in another way by conditionalizing a variable assignment and then using the variable unconditionally:

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

7.2 Syntax of Conditionals

The syntax of a simple conditional with no else is as follows:

```
conditional-directive
text-if-true
endif
```

The *text-if-true* may be any lines of text, to be considered as part of the makefile if the condition is true. If the condition is false, no text is used instead.

The syntax of a complex conditional is as follows:

```
conditional-directive
text-if-true
else
text-if-false
endif
```

If the condition is true, *text-if-true* is used; otherwise, *text-if-false* is used instead. The *text-if-false* can be any number of lines of text.

The syntax of the *conditional-directive* is the same whether the conditional is simple or complex. There are four different directives that test different conditions. Here is a table of them:

`ifeq (`*arg1*`, `*arg2*`)`
`ifeq '`*arg1*`' '`*arg2*`'`
`ifeq "`*arg1*`" "`*arg2*`"`
`ifeq "`*arg1*`" '`*arg2*`'`
`ifeq '`*arg1*`' "`*arg2*`"`

Expand all variable references in *arg1* and *arg2* and compare them. If they are identical, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

Often you want to test if a variable has a non-empty value. When the value results from complex expansions of variables and functions, expansions you would consider empty may actually contain whitespace characters and thus are not seen as empty. However, you can use the `strip` function (see Section 8.2 “Text Functions,” page 72) to avoid interpreting whitespace as a non-empty value. For example:

```
ifeq ($(strip $(foo)),)
text-if-empty
endif
```

will evaluate *text-if-empty* even if the expansion of `$(foo)` contains whitespace characters.

`ifneq (`*arg1*`, `*arg2*`)`
`ifneq '`*arg1*`' '`*arg2*`'`
`ifneq "`*arg1*`" "`*arg2*`"`
`ifneq "`*arg1*`" '`*arg2*`'`
`ifneq '`*arg1*`' "`*arg2*`"`

Expand all variable references in *arg1* and *arg2* and compare them. If they are different, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

`ifdef `*variable-name*

If the variable *variable-name* has a non-empty value, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective. Variables that have never been defined have an empty value.

Note that `ifdef` only tests whether a variable has a value. It does not expand the variable to see if that value is nonempty. Consequently, tests using `ifdef` return true for all definitions except those like `foo =`. To test for an empty value, use `ifeq ($(foo),)`. For example,

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

sets 'frobozz' to 'yes', while:

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

sets 'frobozz' to 'no'.

`ifndef` *variable-name*
: If the variable *variable-name* has an empty value, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

Extra spaces are allowed and ignored at the beginning of the conditional directive line, but a tab is not allowed. (If the line begins with a tab, it will be considered a command for a rule.) Aside from this, extra spaces or tabs may be inserted with no effect anywhere except within the directive name or within an argument. A comment starting with '#' may appear at the end of the line.

The other two directives that play a part in a conditional are `else` and `endif`. Each of these directives is written as one word, with no arguments. Extra spaces are allowed and ignored at the beginning of the line, and spaces or tabs at the end. A comment starting with '#' may appear at the end of the line.

Conditionals affect which lines of the makefile `make` uses. If the condition is true, `make` reads the lines of the *text-if-true* as part of the makefile; if the condition is false, `make` ignores those lines completely. It follows that syntactic units of the makefile, such as rules, may safely be split across the beginning or the end of the conditional.

`make` evaluates conditionals when it reads a makefile. Consequently, you cannot use automatic variables in the tests of conditionals because they are not defined until commands are run (see Section 10.5.3 "Automatic Variables," page 106).

To prevent intolerable confusion, it is not permitted to start a conditional in one makefile and end it in another. However, you may write an

`include` directive within a conditional, provided you do not attempt to terminate the conditional inside the included file.

7.3 Conditionals that Test Flags

You can write a conditional that tests `make` command flags such as '`-t`' by using the variable `MAKEFLAGS` together with the `findstring` function (see Section 8.2 "Functions for String Substitution and Analysis," page 72). This is useful when `touch` is not enough to make a file appear up to date.

The `findstring` function determines whether one string appears as a substring of another. If you want to test for the '`-t`' flag, use '`t`' as the first string and the value of `MAKEFLAGS` as the other.

For example, here is how to arrange to use '`ranlib -t`' to finish marking an archive file up to date:

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        +touch archive.a
        +ranlib -t archive.a
else
        ranlib archive.a
endif
```

The '`+`' prefix marks those command lines as "recursive" so that they will be executed despite use of the '`-t`' flag. See Section 5.6 "Recursive Use of `make`," page 41.

8 Functions for Transforming Text

Functions allow you to do text processing in the makefile to compute the files to operate on or the commands to use. You use a function in a *function call*, where you give the name of the function and some text (the *arguments*) for the function to operate on. The result of the function's processing is substituted into the makefile at the point of the call, just as a variable might be substituted.

8.1 Function Call Syntax

A function call resembles a variable reference. It looks like this:

```
$(function arguments)
```

or like this:

```
${function arguments}
```

Here `function` is a function name; one of a short list of names that are part of `make`. There is no provision for defining new functions.

The `arguments` are the arguments of the function. They are separated from the function name by one or more spaces or tabs, and if there is more than one argument, then they are separated by commas. Such whitespace and commas are not part of an argument's value. The delimiters which you use to surround the function call, whether parentheses or braces, can appear in an argument only in matching pairs; the other kind of delimiters may appear singly. If the arguments themselves contain other function calls or variable references, it is wisest to use the same kind of delimiters for all the references; write '`$(subst a,b,$(x))`', not '`$(subst a,b,${x})`'. This is because it is clearer, and because only one type of delimiter is matched to find the end of the reference.

The text written for each argument is processed by substitution of variables and function calls to produce the argument value, which is the text on which the function acts. The substitution is done in the order in which the arguments appear.

Commas and unmatched parentheses or braces cannot appear in the text of an argument as written; leading spaces cannot appear in the text of the first argument as written. These characters can be put into the argument value by variable substitution. First define variables `comma` and `space` whose values are isolated comma and space characters, then substitute these variables where such characters are wanted, like this:

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now ‘a,b,c’.
```

Here the `subst` function replaces each space with a comma, through the value of `foo`, and substitutes the result.

8.2 Functions for String Substitution and Analysis

Here are some functions that operate on strings:

`$(subst from,to,text)`

Performs a textual replacement on the text *text*: each occurrence of *from* is replaced by *to*. The result is substituted for the function call. For example,

```
$(subst ee,EE,feet on the street)
```

substitutes the string ‘`fEEt on the strEEt`’.

`$(patsubst pattern,replacement,text)`

Finds whitespace-separated words in *text* that match *pattern* and replaces them with *replacement*. Here *pattern* may contain a ‘`%`’ which acts as a wildcard, matching any number of any characters within a word. If *replacement* also contains a ‘`%`’, the ‘`%`’ is replaced by the text that matched the ‘`%`’ in *pattern*.

‘`%`’ characters in `patsubst` function invocations can be quoted with preceding backslashes (‘\’). Backslashes that would otherwise quote ‘`%`’ characters can be quoted with more backslashes. Backslashes that quote ‘`%`’ characters or other backslashes are removed from the pattern before it is compared file names or has a stem substituted into it. Backslashes that are not in danger of quoting ‘`%`’ characters go unmolested. For example, the pattern ‘`the\%weird\\%pattern\\`’ has ‘`the%weird\`’ preceding the operative ‘`%`’ character, and ‘`pattern\\`’ following it. The final two backslashes are left alone because they cannot affect any ‘`%`’ character.

Whitespace between words is folded into single space characters; leading and trailing whitespace is discarded.

For example,

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

produces the value '`x.c.o bar.o`'.

Substitution references (see Section 6.3.1 "Substitution References," page 55) are a simpler way to get the effect of the `patsubst` function:

```
$(var:pattern=replacement)
```

is equivalent to

```
$(patsubst pattern,replacement,$(var))
```

The second shorthand simplifies one of the most common uses of `patsubst`: replacing the suffix at the end of file names.

```
$(var:suffix=replacement)
```

is equivalent to

```
$(patsubst %suffix,%replacement,$(var))
```

For example, you might have a list of object files:

```
objects = foo.o bar.o baz.o
```

To get the list of corresponding source files, you could simply write:

```
$(objects:.o=.c)
```

instead of using the general form:

```
$(patsubst %.o,%.c,$(objects))
```

`$(strip string)`

Removes leading and trailing whitespace from *string* and replaces each internal sequence of one or more whitespace characters with a single space. Thus, '`$(strip a b c )`' results in '`a b c`'.

The function `strip` can be very useful when used in conjunction with conditionals. When comparing something with the empty string '' using `ifeq` or `ifneq`, you usually want a string of just whitespace to match the empty string (see Chapter 7 "Conditionals," page 65).

Thus, the following may fail to have the desired results:

```
.PHONY: all
ifneq   "$(needs_made)" ""
all: $(needs_made)
else
all:;@echo 'Nothing to make!'
endif
```

Replacing the variable reference '`$(needs_made)`' with the function call '`$(strip $(needs_made))`' in the `ifneq` directive would make it more robust.

`$(findstring find,in)`

Searches *in* for an occurrence of *find*. If it occurs, the value is *find*; otherwise, the value is empty. You can use this

function in a conditional to test for the presence of a specific substring in a given string. Thus, the two examples,

```
$(findstring a,a b c)
$(findstring a,b c)
```

produce the values 'a' and '' (the empty string), respectively. See Section 7.3 "Testing Flags," page 69, for a practical application of `findstring`.

`$(filter` *`pattern...`*`,`*`text`*`)`

Removes all whitespace-separated words in *text* that do *not* match any of the *pattern* words, returning only matching words. The patterns are written using '%', just like the patterns used in the `patsubst` function above.

The `filter` function can be used to separate out different types of strings (such as file names) in a variable. For example:

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```

says that 'foo' depends of 'foo.c', 'bar.c', 'baz.s' and 'ugh.h' but only 'foo.c', 'bar.c' and 'baz.s' should be specified in the command to the compiler.

`$(filter-out` *`pattern...`*`,`*`text`*`)`

Removes all whitespace-separated words in *text* that *do* match the *pattern* words, returning only the words that *do not* match. This is the exact opposite of the `filter` function.

For example, given:

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

the following generates a list which contains all the object files not in 'mains':

```
$(filter-out $(mains),$(objects))
```

`$(sort` *`list`*`)`

Sorts the words of *list* in lexical order, removing duplicate words. The output is a list of words separated by single spaces. Thus,

```
$(sort foo bar lose)
```

returns the value 'bar foo lose'.

Incidentally, since `sort` removes duplicate words, you can use it for this purpose even if you don't care about the sort order.

Here is a realistic example of the use of `subst` and `patsubst`. Suppose that a makefile uses the `VPATH` variable to specify a list of directories that

`make` should search for dependency files (see Section 4.3.1 "`VPATH` Search Path for All Dependencies," page 21). This example shows how to tell the C compiler to search for header files in the same list of directories.

The value of `VPATH` is a list of directories separated by colons, such as '`src:../headers`'. First, the `subst` function is used to change the colons to spaces:

```
$(subst :, ,$(VPATH))
```

This produces '`src ../headers`'. Then `patsubst` is used to turn each directory name into a '`-I`' flag. These can be added to the value of the variable `CFLAGS`, which is passed automatically to the C compiler, like this:

```
override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))
```

The effect is to append the text '`-Isrc -I../headers`' to the previously given value of `CFLAGS`. The `override` directive is used so that the new value is assigned even if the previous value of `CFLAGS` was specified with a command argument (see Section 6.7 "The `override` Directive," page 61).

8.3 Functions for File Names

Several of the built-in expansion functions relate specifically to taking apart file names or lists of file names.

Each of the following functions performs a specific transformation on a file name. The argument of the function is regarded as a series of file names, separated by whitespace. (Leading and trailing whitespace is ignored.) Each file name in the series is transformed in the same way and the results are concatenated with single spaces between them.

`$(dir` *`names`*`...)`

Extracts the directory-part of each file name in *names*. The directory-part of the file name is everything up through (and including) the last slash in it. If the file name contains no slash, the directory part is the string '`./`'. For example,

```
$(dir src/foo.c hacks)
```

produces the result '`src/ ./`'.

`$(notdir` *`names`*`...)`

Extracts all but the directory-part of each file name in *names*. If the file name contains no slash, it is left unchanged. Otherwise, everything through the last slash is removed from it.

A file name that ends with a slash becomes an empty string. This is unfortunate, because it means that the result does

not always have the same number of whitespace-separated file names as the argument had; but we do not see any other valid alternative.

For example,

```
$(notdir src/foo.c hacks)
```

produces the result '`foo.c hacks`'.

`$(suffix names...)`
Extracts the suffix of each file name in *names*. If the file name contains a period, the suffix is everything starting with the last period. Otherwise, the suffix is the empty string. This frequently means that the result will be empty when *names* is not, and if *names* contains multiple file names, the result may contain fewer file names.

For example,

```
$(suffix src/foo.c hacks)
```

produces the result '`.c`'.

`$(basename names...)`
Extracts all but the suffix of each file name in *names*. If the file name contains a period, the basename is everything starting up to (and not including) the last period. Otherwise, the basename is the entire file name. For example,

```
$(basename src/foo.c hacks)
```

produces the result '`src/foo hacks`'.

`$(addsuffix suffix,names...)`
The argument *names* is regarded as a series of names, separated by whitespace; *suffix* is used as a unit. The value of *suffix* is appended to the end of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addsuffix .c,foo bar)
```

produces the result '`foo.c bar.c`'.

`$(addprefix prefix,names...)`
The argument *names* is regarded as a series of names, separated by whitespace; *prefix* is used as a unit. The value of *prefix* is prepended to the front of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addprefix src/,foo bar)
```

produces the result '`src/foo src/bar`'.

`$(join list1,list2)`
: Concatenates the two arguments word by word: the two first words (one from each argument) concatenated form the first word of the result, the two second words form the second word of the result, and so on. So the *n*th word of the result comes from the *n*th word of each argument. If one argument has more words that the other, the extra words are copied unchanged into the result.

  For example, '`$(join a b,.c .o)`' produces '`a.c b.o`'.

  Whitespace between the words in the lists is not preserved; it is replaced with a single space.

  This function can merge the results of the `dir` and `notdir` functions, to produce the original list of files which was given to those two functions.

`$(word n,text)`
: Returns the *n*th word of *text*. The legitimate values of *n* start from 1. If *n* is bigger than the number of words in *text*, the value is empty. For example,

  ```
  $(word 2, foo bar baz)
  ```

  returns '`bar`'.

`$(words text)`
: Returns the number of words in *text*. Thus, the last word of *text* is `$(word $(words text),text)`.

`$(firstword names...)`
: The argument *names* is regarded as a series of names, separated by whitespace. The value is the first name in the series. The rest of the names are ignored.

  For example,

  ```
  $(firstword foo bar)
  ```

  produces the result '`foo`'. Although `$(firstword text)` is the same as `$(word 1,text)`, the `firstword` function is retained for its simplicity.

`$(wildcard pattern)`
: The argument *pattern* is a file name pattern, typically containing wildcard characters (as in shell file name patterns). The result of `wildcard` is a space-separated list of the names of existing files that match the pattern. See Section 4.2 "Using Wildcard Characters in File Names," page 18.

8.4 The foreach Function

The foreach function is very different from other functions. It causes one piece of text to be used repeatedly, each time with a different substitution performed on it. It resembles the for command in the shell sh and the foreach command in the C-shell csh.

The syntax of the foreach function is:

```
$(foreach var,list,text)
```

The first two arguments, *var* and *list*, are expanded before anything else is done; note that the last argument, *text*, is **not** expanded at the same time. Then for each word of the expanded value of *list*, the variable named by the expanded value of *var* is set to that word, and *text* is expanded. Presumably *text* contains references to that variable, so its expansion will be different each time.

The result is that *text* is expanded as many times as there are whitespace-separated words in *list*. The multiple expansions of *text* are concatenated, with spaces between them, to make the result of foreach.

This simple example sets the variable 'files' to the list of all files in the directories in the list 'dirs':

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

Here *text* is '$(wildcard $(dir)/*)'. The first repetition finds the value 'a' for dir, so it produces the same result as '$(wildcard a/*)'; the second repetition produces the result of '$(wildcard b/*)'; and the third, that of '$(wildcard c/*)'.

This example has the same result (except for setting 'dirs') as the following example:

```
files := $(wildcard a/* b/* c/* d/*)
```

When *text* is complicated, you can improve readability by giving it a name, with an additional variable:

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

Here we use the variable find_files this way. We use plain '=' to define a recursively-expanding variable, so that its value contains an actual function call to be reexpanded under the control of foreach; a simply-expanded variable would not do, since wildcard would be called only once at the time of defining find_files.

The foreach function has no permanent effect on the variable *var*; its value and flavor after the foreach function call are the same as they were beforehand. The other values which are taken from *list* are in

effect only temporarily, during the execution of `foreach`. The variable *`var`* is a simply-expanded variable during the execution of `foreach`. If *`var`* was undefined before the `foreach` function call, it is undefined after the call. See Section 6.2 "The Two Flavors of Variables," page 52.

You must take care when using complex variable expressions that result in variable names because many strange things are valid variable names, but are probably not what you intended. For example,

```
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
```

might be useful if the value of `find_files` references the variable whose name is '`Esta escrito en espanol!`' (es un nombre bastante largo, no?), but it is more likely to be a mistake.

8.5 The `origin` Function

The `origin` function is unlike most other functions in that it does not operate on the values of variables; it tells you something *about* a variable. Specifically, it tells you where it came from.

The syntax of the `origin` function is:

```
$(origin variable)
```

Note that *`variable`* is the *name* of a variable to inquire about; not a *reference* to that variable. Therefore you would not normally use a '`$`' or parentheses when writing it. (You can, however, use a variable reference in the name if you want the name not to be a constant.)

The result of this function is a string telling you how the variable *`variable`* was defined:

'`undefined`'
: if *`variable`* was never defined.

'`default`'
: if *`variable`* has a default definition, as is usual with `CC` and so on. See Section 10.3 "Variables Used by Implicit Rules," page 100. Note that if you have redefined a default variable, the `origin` function will return the origin of the later definition.

'`environment`'
: if *`variable`* was defined as an environment variable and the '`-e`' option is *not* turned on (see Section 9.7 "Summary of Options," page 89).

'`environment override`'
: if *`variable`* was defined as an environment variable and the '`-e`' option *is* turned on (see Section 9.7 "Summary of Options," page 89).

'file'
: if *variable* was defined in a makefile.

'command line'
: if *variable* was defined on the command line.

'override'
: if *variable* was defined with an override directive in a makefile (see Section 6.7 "The override Directive," page 61).

'automatic'
: if *variable* is an automatic variable defined for the execution of the commands for each rule (see Section 10.5.3 "Automatic Variables," page 106).

This information is primarily useful (other than for your curiosity) to determine if you want to believe the value of a variable. For example, suppose you have a makefile 'foo' that includes another makefile 'bar'. You want a variable bletch to be defined in 'bar' if you run the command 'make -f bar', even if the environment contains a definition of bletch. However, if 'foo' defined bletch before including 'bar', you do not want to override that definition. This could be done by using an override directive in 'foo', giving that definition precedence over the later definition in 'bar'; unfortunately, the override directive would also override any command line definitions. So, 'bar' could include:

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

If bletch has been defined from the environment, this will redefine it.

If you want to override a previous definition of bletch if it came from the environment, even under '-e', you could instead write:

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

Here the redefinition takes place if '$(origin bletch)' returns either 'environment' or 'environment override'. See Section 8.2 "Functions for String Substitution and Analysis," page 72.

8.6 The shell Function

The shell function is unlike any other function except the wildcard function (see Section 4.2.3 "The Function wildcard," page 20) in that it communicates with the world outside of make.

The `shell` function performs the same function that backquotes ('`') perform in most shells: it does *command expansion*. This means that it takes an argument that is a shell command and returns the output of the command. The only processing `make` does on the result, before substituting it into the surrounding text, is to convert newlines to spaces.

The commands run by calls to the `shell` function are run when the function calls are expanded. In most cases, this is when the makefile is read in. The exception is that function calls in the commands of the rules are expanded when the commands are run, and this applies to `shell` function calls like all others.

Here are some examples of the use of the `shell` function:

```
contents := $(shell cat foo)
```

sets `contents` to the contents of the file '`foo`', with a space (rather than a newline) separating each line.

```
files := $(shell echo *.c)
```

sets `files` to the expansion of '`*.c`'. Unless `make` is using a very strange shell, this has the same result as '`$(wildcard *.c)`'.

9 How to Run make

A makefile that says how to recompile a program can be used in more than one way. The simplest use is to recompile every file that is out of date. Usually, makefiles are written so that if you run make with no arguments, it does just that.

But you might want to update only some of the files; you might want to use a different compiler or different compiler options; you might want just to find out which files are out of date without changing them.

By giving arguments when you run make, you can do any of these things and many others.

The exit status of make is always one of three values:

0 The exit status is zero if make is successful.

2 The exit status is two if make encounters any errors. It will print messages describing the particular errors.

1 The exit status is one if you use the '-q' flag and make determines that some target is not already up to date. See Section 9.3 "Instead of Executing the Commands," page 85.

9.1 Arguments to Specify the Makefile

The way to specify the name of the makefile is with the '-f' or '--file' option ('--makefile' also works). For example, '-f altmake' says to use the file 'altmake' as the makefile.

If you use the '-f' flag several times and follow each '-f' with an argument, all the specified files are used jointly as makefiles.

If you do not use the '-f' or '--file' flag, the default is to try 'GNUmakefile', 'makefile', and 'Makefile', in that order, and use the first of these three which exists or can be made (see Chapter 3 "Writing Makefiles," page 11).

9.2 Arguments to Specify the Goals

The *goals* are the targets that make should strive ultimately to update. Other targets are updated as well if they appear as dependencies of goals, or dependencies of dependencies of goals, etc.

By default, the goal is the first target in the makefile (not counting targets that start with a period). Therefore, makefiles are usually written so that the first target is for compiling the entire program or programs

they describe. If the first rule in the makefile has several targets, only the first target in the rule becomes the default goal, not the whole list.

You can specify a different goal or goals with arguments to make. Use the name of the goal as an argument. If you specify several goals, make processes each of them in turn, in the order you name them.

Any target in the makefile may be specified as a goal (unless it starts with '-' or contains an '=', in which case it will be parsed as a switch or variable definition, respectively). Even targets not in the makefile may be specified, if make can find implicit rules that say how to make them.

One use of specifying a goal is if you want to compile only a part of the program, or only one of several programs. Specify as a goal each file that you wish to remake. For example, consider a directory containing several programs, with a makefile that starts like this:

```
.PHONY: all
all: size nm ld ar as
```

If you are working on the program size, you might want to say 'make size' so that only the files of that program are recompiled.

Another use of specifying a goal is to make files that are not normally made. For example, there may be a file of debugging output, or a version of the program that is compiled specially for testing, which has a rule in the makefile but is not a dependency of the default goal.

Another use of specifying a goal is to run the commands associated with a phony target (see Section 4.4 "Phony Targets," page 24) or empty target (see Section 4.6 "Empty Target Files to Record Events," page 26). Many makefiles contain a phony target named 'clean' which deletes everything except source files. Naturally, this is done only if you request it explicitly with 'make clean'. Following is a list of typical phony and empty target names. See Section 14.3 "Standard Targets," page 126, for a detailed list of all the standard target names which GNU software packages use.

'all' Make all the top-level targets the makefile knows about.

'clean' Delete all files that are normally created by running make.

'mostlyclean'
: Like 'clean', but may refrain from deleting a few files that people normally don't want to recompile. For example, the 'mostlyclean' target for GCC does not delete 'libgcc.a', because recompiling it is rarely necessary and takes a lot of time.

'distclean'
'realclean'
'clobber' Any of these targets might be defined to delete *more* files than 'clean' does. For example, this would delete configuration files or links that you would normally create as preparation for compilation, even if the makefile itself cannot create these files.

'install' Copy the executable file into a directory that users typically search for commands; copy any auxiliary files that the executable uses into the directories where it will look for them.

'print' Print listings of the source files that have changed.

'tar' Create a tar file of the source files.

'shar' Create a shell archive (shar file) of the source files.

'dist' Create a distribution file of the source files. This might be a tar file, or a shar file, or a compressed version of one of the above, or even more than one of the above.

'TAGS' Update a tags table for this program.

'check'
'test' Perform self tests on the program this makefile builds.

9.3 Instead of Executing the Commands

The makefile tells make how to tell whether a target is up to date, and how to update each target. But updating the targets is not always what you want. Certain options specify other activities for make.

'-n'
'--just-print'
'--dry-run'
'--recon'

"No-op". The activity is to print what commands would be used to make the targets up to date, but not actually execute them.

'-t'
'--touch'

"Touch". The activity is to mark the targets as up to date without actually changing them. In other words, make pretends to compile the targets but does not really change their contents.

'-q'
'--question'
: "Question". The activity is to find out silently whether the targets are up to date already; but execute no commands in either case. In other words, neither compilation nor output will occur.

'-W *file*'
'--what-if=*file*'
'--assume-new=*file*'
'--new-file=*file*'
: "What if". Each '-W' flag is followed by a file name. The given files' modification times are recorded by make as being the present time, although the actual modification times remain the same. You can use the '-W' flag in conjunction with the '-n' flag to see what would happen if you were to modify specific files.

With the '-n' flag, make prints the commands that it would normally execute but does not execute them.

With the '-t' flag, make ignores the commands in the rules and uses (in effect) the command touch for each target that needs to be remade. The touch command is also printed, unless '-s' or .SILENT is used. For speed, make does not actually invoke the program touch. It does the work directly.

With the '-q' flag, make prints nothing and executes no commands, but the exit status code it returns is zero if and only if the targets to be considered are already up to date. If the exit status is one, then some updating needs to be done. If make encounters an error, the exit status is two, so you can distinguish an error from a target that is not up to date.

It is an error to use more than one of these three flags in the same invocation of make.

The '-n', '-t', and '-q' options do not affect command lines that begin with '+' characters or contain the strings '$(MAKE)' or '${MAKE}'. Note that only the line containing the '+' character or the strings '$(MAKE)' or '${MAKE}' is run regardless of these options. Other lines in the same rule are not run unless they too begin with '+' or contain '$(MAKE)' or '${MAKE}' (See Section 5.6.1 "How the MAKE Variable Works," page 42.)

The '-W' flag provides two features:

- If you also use the '-n' or '-q' flag, you can see what make would do if you were to modify some files.
- Without the '-n' or '-q' flag, when make is actually executing commands, the '-W' flag can direct make to act as if some files had been modified, without actually modifying the files.

Note that the options '-p' and '-v' allow you to obtain other information about make or about the makefiles in use (see Section 9.7 "Summary of Options," page 89).

9.4 Avoiding Recompilation of Some Files

Sometimes you may have changed a source file but you do not want to recompile all the files that depend on it. For example, suppose you add a macro or a declaration to a header file that many other files depend on. Being conservative, make assumes that any change in the header file requires recompilation of all dependent files, but you know that they do not need to be recompiled and you would rather not waste the time waiting for them to compile.

If you anticipate the problem before changing the header file, you can use the '-t' flag. This flag tells make not to run the commands in the rules, but rather to mark the target up to date by changing its last-modification date. You would follow this procedure:

1. Use the command 'make' to recompile the source files that really need recompilation.
2. Make the changes in the header files.
3. Use the command 'make -t' to mark all the object files as up to date. The next time you run make, the changes in the header files will not cause any recompilation.

If you have already changed the header file at a time when some files do need recompilation, it is too late to do this. Instead, you can use the '-o *file*' flag, which marks a specified file as "old" (see Section 9.7 "Summary of Options," page 89). This means that the file itself will not be remade, and nothing else will be remade on its account. Follow this procedure:

1. Recompile the source files that need compilation for reasons independent of the particular header file, with 'make -o *headerfile*'. If several header files are involved, use a separate '-o' option for each header file.
2. Touch all the object files with 'make -t'.

9.5 Overriding Variables

An argument that contains '=' specifies the value of a variable: '*v*=*x*' sets the value of the variable *v* to *x*. If you specify a value in this way, all ordinary assignments of the same variable in the makefile are ignored; we say they have been *overridden* by the command line argument.

The most common way to use this facility is to pass extra flags to compilers. For example, in a properly written makefile, the variable CFLAGS is included in each command that runs the C compiler, so a file 'foo.c' would be compiled something like this:

```
cc -c $(CFLAGS) foo.c
```

Thus, whatever value you set for CFLAGS affects each compilation that occurs. The makefile probably specifies the usual value for CFLAGS, like this:

```
CFLAGS=-g
```

Each time you run make, you can override this value if you wish. For example, if you say 'make CFLAGS='-g -O'', each C compilation will be done with 'cc -c -g -O'. (This illustrates how you can use quoting in the shell to enclose spaces and other special characters in the value of a variable when you override it.)

The variable CFLAGS is only one of many standard variables that exist just so that you can change them this way. See Section 10.3 "Variables Used by Implicit Rules," page 100, for a complete list.

You can also program the makefile to look at additional variables of your own, giving the user the ability to control other aspects of how the makefile works by changing the variables.

When you override a variable with a command argument, you can define either a recursively-expanded variable or a simply-expanded variable. The examples shown above make a recursively-expanded variable; to make a simply-expanded variable, write ':=' instead of '='. But, unless you want to include a variable reference or function call in the *value* that you specify, it makes no difference which kind of variable you create.

There is one way that the makefile can change a variable that you have overridden. This is to use the override directive, which is a line that looks like this: 'override *variable* = *value*' (see Section 6.7 "The override Directive," page 61).

9.6 Testing the Compilation of a Program

Normally, when an error happens in executing a shell command, make gives up immediately, returning a nonzero status. No further commands are executed for any target. The error implies that the goal cannot be correctly remade, and make reports this as soon as it knows.

When you are compiling a program that you have just changed, this is not what you want. Instead, you would rather that make try compiling every file that can be tried, to show you as many compilation errors as possible.

On these occasions, you should use the '-k' or '--keep-going' flag. This tells make to continue to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, 'make -k' will continue compiling other object files even though it already knows that linking them will be impossible. In addition to continuing after failed shell commands, 'make -k' will continue as much as possible after discovering that it does not know how to make a target or dependency file. This will always cause an error message, but without '-k', it is a fatal error (see Section 9.7 "Summary of Options," page 89).

The usual behavior of make assumes that your purpose is to get the goals up to date; once make learns that this is impossible, it might as well report the failure immediately. The '-k' flag says that the real purpose is to test as much as possible of the changes made in the program, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs' M-x compile command passes the '-k' flag by default.

9.7 Summary of Options

Here is a table of all the options make understands:

'-b'
'-m'
: These options are ignored for compatibility with other versions of make.

'-C *dir*'
'--directory=*dir*'
: Change to directory *dir* before reading the makefiles. If multiple '-C' options are specified, each is interpreted relative to the previous one: '-C / -C etc' is equivalent to '-C /etc'. This is typically used with recursive invocations of make (see Section 5.6 "Recursive Use of make," page 41).

'-d'
'--debug'
: Print debugging information in addition to normal processing. The debugging information says which files are being considered for remaking, which file-times are being compared and with what results, which files actually need to be remade, which implicit rules are considered and which are applied—everything interesting about how make decides what to do.

'-e'
'--environment-overrides'
Give variables taken from the environment precedence over variables from makefiles. See Section 6.9 "Variables from the Environment," page 62.

'-f *file*'
'--file=*file*'
'--makefile=*file*'
Read the file named *file* as a makefile. See Chapter 3 "Writing Makefiles," page 11.

'-h'
'--help'

Remind you of the options that make understands and then exit.

'-i'
'--ignore-errors'
Ignore all errors in commands executed to remake files. See Section 5.4 "Errors in Commands," page 40.

'-I *dir*'
'--include-dir=*dir*'
Specifies a directory *dir* to search for included makefiles. See Section 3.3 "Including Other Makefiles," page 12. If several '-I' options are used to specify several directories, the directories are searched in the order specified.

'-j [*jobs*]'
'--jobs=[*jobs*]'
Specifies the number of jobs (commands) to run simultaneously. With no argument, make runs as many jobs simultaneously as possible. If there is more than one '-j' option, the last one is effective. See Section 5.3 "Parallel Execution," page 38, for more information on how commands are run.

'-k'
'--keep-going'
Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same. See Section 9.6 "Testing the Compilation of a Program," page 88.

'-l [*load*]'
'--load-average[=*load*]'
'--max-load[=*load*]'
Specifies that no new jobs (commands) should be started if there are other jobs running and the load average is at least *load* (a floating-point number). With no argument, removes a previous load limit. See Section 5.3 "Parallel Execution," page 38.

'-n'
'--just-print'
'--dry-run'
'--recon'
Print the commands that would be executed, but do not execute them. See Section 9.3 "Instead of Executing the Commands," page 85.

'-o *file*'
'--old-file=*file*'
'--assume-old=*file*'
Do not remake the file *file* even if it is older than its dependencies, and do not remake anything on account of changes in *file*. Essentially the file is treated as very old and its rules are ignored. See Section 9.4 "Avoiding Recompilation of Some Files," page 87.

'-p'
'--print-data-base'
Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as otherwise specified. This also prints the version information given by the '-v' switch (see below). To print the data base without trying to remake any files, use 'make -p -f /dev/null'.

'-q'
'--question'
"Question mode". Do not run any commands, or print anything; just return an exit status that is zero if the specified targets are already up to date, one if any remaking is required, or two if an error is encountered. See Section 9.3 "Instead of Executing the Commands," page 85.

'-r'
'--no-builtin-rules'
Eliminate use of the built-in implicit rules (see Chapter 10 "Using Implicit Rules," page 95). You can still define your own by writing pattern rules (see Section 10.5 "Defining and

Redefining Pattern Rules," page 104). The '-r' option also clears out the default list of suffixes for suffix rules (see Section 10.7 "Old-Fashioned Suffix Rules," page 111). But you can still define your own suffixes with a rule for .SUFFIXES, and then define your own suffix rules.

'-s'
'--silent'
'--quiet'

Silent operation; do not print the commands as they are executed. See Section 5.1 "Command Echoing," page 37.

'-S'
'--no-keep-going'
'--stop'

Cancel the effect of the '-k' option. This is never necessary except in a recursive make where '-k' might be inherited from the top-level make via MAKEFLAGS (see Section 5.6 "Recursive Use of make," page 41) or if you set '-k' in MAKEFLAGS in your environment.

'-t'
'--touch'

Touch files (mark them up to date without really changing them) instead of running their commands. This is used to pretend that the commands were done, in order to fool future invocations of make. See Section 9.3 "Instead of Executing the Commands," page 85.

'-v'
'--version'

Print the version of the make program plus a copyright, a list of authors, and a notice that there is no warranty; then exit.

'-w'
'--print-directory'

Print a message containing the working directory both before and after executing the makefile. This may be useful for tracking down errors from complicated nests of recursive make commands. See Section 5.6 "Recursive Use of make," page 41. (In practice, you rarely need to specify this option since 'make' does it for you; see Section 5.6.4 "The '--print-directory' Option," page 47.)

'--no-print-directory'

Disable printing of the working directory under -w. This option is useful when -w is turned on automatically, but you

do not want to see the extra messages. See Section 5.6.4 "The '`--print-directory`' Option," page 47.

'`-W` *`file`*'
'`--what-if=`*`file`*'
'`--new-file=`*`file`*'
'`--assume-new=`*`file`*'

Pretend that the target *`file`* has just been modified. When used with the '`-n`' flag, this shows you what would happen if you were to modify that file. Without '`-n`', it is almost the same as running a `touch` command on the given file before running `make`, except that the modification time is changed only in the imagination of `make`. See Section 9.3 "Instead of Executing the Commands," page 85.

'`--warn-undefined-variables`'

Issue a warning message whenever `make` sees a reference to an undefined variable. This can be helpful when you are trying to debug makefiles which use variables in complex ways.

10 Using Implicit Rules

Certain standard ways of remaking target files are used very often. For example, one customary way to make an object file is from a C source file using the C compiler, `cc`.

Implicit rules tell `make` how to use customary techniques so that you do not have to specify them in detail when you want to use them. For example, there is an implicit rule for C compilation. File names determine which implicit rules are run. For example, C compilation typically takes a '`.c`' file and makes a '`.o`' file. So `make` applies the implicit rule for C compilation when it sees this combination of file name endings.

A chain of implicit rules can apply in sequence; for example, `make` will remake a '`.o`' file from a '`.y`' file by way of a '`.c`' file. See Section 10.4 "Chains of Implicit Rules," page 103.

The built-in implicit rules use several variables in their commands so that, by changing the values of the variables, you can change the way the implicit rule works. For example, the variable `CFLAGS` controls the flags given to the C compiler by the implicit rule for C compilation. See Section 10.3 "Variables Used by Implicit Rules," page 100.

You can define your own implicit rules by writing *pattern rules*. See Section 10.5 "Defining and Redefining Pattern Rules," page 104.

Suffix rules are a more limited way to define implicit rules. Pattern rules are more general and clearer, but suffix rules are retained for compatibility. See Section 10.7 "Old-Fashioned Suffix Rules," page 111.

10.1 Using Implicit Rules

To allow `make` to find a customary method for updating a target file, all you have to do is refrain from specifying commands yourself. Either write a rule with no command lines, or don't write a rule at all. Then `make` will figure out which implicit rule to use based on which kind of source file exists or can be made.

For example, suppose the makefile looks like this:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Because you mention '`foo.o`' but do not give a rule for it, `make` will automatically look for an implicit rule that tells how to update it. This happens whether or not the file '`foo.o`' currently exists.

If an implicit rule is found, it can supply both commands and one or more dependencies (the source files). You would want to write a rule for '`foo.o`' with no command lines if you need to specify additional dependencies, such as header files, that the implicit rule cannot supply.

Each implicit rule has a target pattern and dependency patterns. There may be many implicit rules with the same target pattern. For example, numerous rules make '.o' files: one, from a '.c' file with the C compiler; another, from a '.p' file with the Pascal compiler; and so on. The rule that actually applies is the one whose dependencies exist or can be made. So, if you have a file 'foo.c', make will run the C compiler; otherwise, if you have a file 'foo.p', make will run the Pascal compiler; and so on.

Of course, when you write the makefile, you know which implicit rule you want make to use, and you know it will choose that one because you know which possible dependency files are supposed to exist. See Section 10.2 "Catalogue of Implicit Rules," page 96, for a catalogue of all the predefined implicit rules.

Above, we said an implicit rule applies if the required dependencies "exist or can be made". A file "can be made" if it is mentioned explicitly in the makefile as a target or a dependency, or if an implicit rule can be recursively found for how to make it. When an implicit dependency is the result of another implicit rule, we say that *chaining* is occurring. See Section 10.4 "Chains of Implicit Rules," page 103.

In general, make searches for an implicit rule for each target, and for each double-colon rule, that has no commands. A file that is mentioned only as a dependency is considered a target whose rule specifies nothing, so implicit rule search happens for it. See Section 10.8 "Implicit Rule Search Algorithm," page 113, for the details of how the search is done.

Note that explicit dependencies do not influence implicit rule search. For example, consider this explicit rule:

```
foo.o: foo.p
```

The dependency on 'foo.p' does not necessarily mean that make will remake 'foo.o' according to the implicit rule to make an object file, a '.o' file, from a Pascal source file, a '.p' file. For example, if 'foo.c' also exists, the implicit rule to make an object file from a C source file is used instead, because it appears before the Pascal rule in the list of predefined implicit rules (see Section 10.2 "Catalogue of Implicit Rules," page 96).

If you do not want an implicit rule to be used for a target that has no commands, you can give that target empty commands by writing a semicolon (see Section 5.8 "Defining Empty Commands," page 48).

10.2 Catalogue of Implicit Rules

Here is a catalogue of predefined implicit rules which are always available unless the makefile explicitly overrides or cancels them. See Section 10.5.6 "Canceling Implicit Rules," page 110, for infor-

mation on canceling or overriding an implicit rule. The '-r' or '--no-builtin-rules' option cancels all predefined rules.

Not all of these rules will always be defined, even when the '-r' option is not given. Many of the predefined implicit rules are implemented in make as suffix rules, so which ones will be defined depends on the *suffix list* (the list of dependencies of the special target .SUFFIXES). The default suffix list is: .out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el. All of the implicit rules described below whose dependencies have one of these suffixes are actually suffix rules. If you modify the suffix list, the only predefined suffix rules in effect will be those named by one or two of the suffixes that are on the list you specify; rules whose suffixes fail to be on the list are disabled. See Section 10.7 "Old-Fashioned Suffix Rules," page 111, for full details on suffix rules.

Compiling C programs
: '*n*.o' is made automatically from '*n*.c' with a command of the form '$(CC) -c $(CPPFLAGS) $(CFLAGS)'.

Compiling C++ programs
: '*n*.o' is made automatically from '*n*.cc' or '*n*.C' with a command of the form '$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)'. We encourage you to use the suffix '.cc' for C++ source files instead of '.C'.

Compiling Pascal programs
: '*n*.o' is made automatically from '*n*.p' with the command '$(PC) -c $(PFLAGS)'.

Compiling Fortran and Ratfor programs
: '*n*.o' is made automatically from '*n*.r', '*n*.F' or '*n*.f' by running the Fortran compiler. The precise command used is as follows:

  '.f' '$(FC) -c $(FFLAGS)'.

  '.F' '$(FC) -c $(FFLAGS) $(CPPFLAGS)'.

  '.r' '$(FC) -c $(FFLAGS) $(RFLAGS)'.

Preprocessing Fortran and Ratfor programs
: '*n*.f' is made automatically from '*n*.r' or '*n*.F'. This rule runs just the preprocessor to convert a Ratfor or preprocessable Fortran program into a strict Fortran program. The precise command used is as follows:

  '.F' '$(FC) -F $(CPPFLAGS) $(FFLAGS)'.

  '.r' '$(FC) -F $(FFLAGS) $(RFLAGS)'.

Compiling Modula-2 programs

'*n*.sym' is made from '*n*.def' with a command of the form '$(M2C) $(M2FLAGS) $(DEFFLAGS)'. '*n*.o' is made from '*n*.mod'; the form is: '$(M2C) $(M2FLAGS) $(MODFLAGS)'.

Assembling and preprocessing assembler programs

'*n*.o' is made automatically from '*n*.s' by running the assembler, as. The precise command is '$(AS) $(ASFLAGS)'.

'*n*.s' is made automatically from '*n*.S' by running the C preprocessor, cpp. The precise command is '$(CPP) $(CPPFLAGS)'.

Linking a single object file

'*n*' is made automatically from '*n*.o' by running the linker (usually called ld) via the C compiler. The precise command used is '$(CC) $(LDFLAGS) *n*.o $(LOADLIBES)'.

This rule does the right thing for a simple program with only one source file. It will also do the right thing if there are multiple object files (presumably coming from various other source files), one of which has a name matching that of the executable file. Thus,

```
x: y.o z.o
```

when 'x.c', 'y.c' and 'z.c' all exist will execute:

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

In more complicated cases, such as when there is no object file whose name derives from the executable file name, you must write an explicit command for linking.

Each kind of file automatically made into '.o' object files will be automatically linked by using the compiler ('$(CC)', '$(FC)' or '$(PC)'; the C compiler '$(CC)' is used to assemble '.s' files) without the '-c' option. This could be done by using the '.o' object files as intermediates, but it is faster to do the compiling and linking in one step, so that's how it's done.

Yacc for C programs

'*n*.c' is made automatically from '*n*.y' by running Yacc with the command '$(YACC) $(YFLAGS)'.

Lex for C programs

'*n*.c' is made automatically from '*n*.l' by by running Lex. The actual command is '$(LEX) $(LFLAGS)'.

Lex for Ratfor programs

'*n*.r' is made automatically from '*n*.l' by by running Lex. The actual command is '$(LEX) $(LFLAGS)'.

The convention of using the same suffix '.l' for all Lex files regardless of whether they produce C code or Ratfor code makes it impossible for make to determine automatically which of the two languages you are using in any particular case. If make is called upon to remake an object file from a '.l' file, it must guess which compiler to use. It will guess the C compiler, because that is more common. If you are using Ratfor, make sure make knows this by mentioning '*n*.r' in the makefile. Or, if you are using Ratfor exclusively, with no C files, remove '.c' from the list of implicit rule suffixes with:

```
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
```

Making Lint Libraries from C, Yacc, or Lex programs

'*n*.ln' is made from '*n*.c' by running lint. The precise command is '$(LINT) $(LINTFLAGS) $(CPPFLAGS) -i'. The same command is used on the C code produced from '*n*.y' or '*n*.l'.

TEX and Web

'*n*.dvi' is made from '*n*.tex' with the command '$(TEX)'. '*n*.tex' is made from '*n*.web' with '$(WEAVE)', or from '*n*.w' (and from '*n*.ch' if it exists or can be made) with '$(CWEAVE)'. '*n*.p' is made from '*n*.web' with '$(TANGLE)' and '*n*.c' is made from '*n*.w' (and from '*n*.ch' if it exists or can be made) with '$(CTANGLE)'.

Texinfo and Info

Use the command '$(TEXI2DVI) $(TEXI2DVI_FLAGS)' to make '*n*.dvi' from either '*n*.texinfo', '*n*.texi', or '*n*.txinfo'. Use the command '$(MAKEINFO) $(MAKEINFO_FLAGS)' to make '*n*.info' from either '*n*.texinfo', '*n*.texi', or '*n*.txinfo'.

RCS

Any file '*n*' is extracted if necessary from an RCS file named either '*n*,v' or 'RCS/*n*,v'. The precise command used is '$(CO) $(COFLAGS)'. '*n*' will not be extracted from RCS if it already exists, even if the RCS file is newer. The rules for RCS are terminal (see Section 10.5.5 "Match-Anything Pattern Rules," page 109), so RCS files cannot be generated from another source; they must actually exist.

SCCS

Any file '*n*' is extracted if necessary from an SCCS file named either 's.*n*' or 'SCCS/s.*n*'. The precise command used is '$(GET) $(GFLAGS)'. The rules for SCCS are terminal (see Section 10.5.5 "Match-Anything Pattern Rules," page 109), so SCCS files cannot be generated from another source; they must actually exist.

For the benefit of SCCS, a file '*n*' is copied from '*n*.sh' and made executable (by everyone). This is for shell scripts that are checked into SCCS. Since RCS preserves the execution permission of a file, you do not need to use this feature with RCS.

We recommend that you avoid using of SCCS. RCS is widely held to be superior, and is also free. By choosing free software in place of comparable (or inferior) proprietary software, you support the free software movement.

Usually, you want to change only the variables listed in the table above, which are documented in the following section.

However, the commands in built-in implicit rules actually use variables such as COMPILE.c, LINK.p, and PREPROCESS.S, whose values contain the commands listed above.

make follows the convention that the rule to compile a '.*x*' source file uses the variable COMPILE.*x*. Similarly, the rule to produce an executable from a '.*x*' file uses LINK.*x*; and the rule to preprocess a '.*x*' file uses PREPROCESS.*x*.

Every rule that produces an object file uses the variable OUTPUT_OPTION. make defines this variable either to contain '-o $@', or to be empty, depending on a compile-time option. You need the '-o' option to ensure that the output goes into the right file when the source file is in a different directory, as when using VPATH (see Section 4.3 "Directory Search," page 20). However, compilers on some systems do not accept a '-o' switch for object files. If you use such a system, and use VPATH, some compilations will put their output in the wrong place. A possible workaround for this problem is to give OUTPUT_OPTION the value '; mv $*.o $@'.

10.3 Variables Used by Implicit Rules

The commands in built-in implicit rules make liberal use of certain predefined variables. You can alter these variables in the makefile, with arguments to make, or in the environment to alter how the implicit rules work without redefining the rules themselves.

For example, the command used to compile a C source file actually says '`$(CC) -c $(CFLAGS) $(CPPFLAGS)`'. The default values of the variables used are '`cc`' and nothing, resulting in the command '`cc -c`'. By redefining '`CC`' to '`ncc`', you could cause '`ncc`' to be used for all C compilations performed by the implicit rule. By redefining '`CFLAGS`' to be '`-g`', you could pass the '`-g`' option to each compilation. *All* implicit rules that do C compilation use '`$(CC)`' to get the program name for the compiler and *all* include '`$(CFLAGS)`' among the arguments given to the compiler.

The variables used in implicit rules fall into two classes: those that are names of programs (like `CC`) and those that contain arguments for the programs (like `CFLAGS`). (The "name of a program" may also contain some command arguments, but it must start with an actual executable program name.) If a variable value contains more than one argument, separate them with spaces.

Here is a table of variables used as names of programs in built-in rules:

`AR` Archive-maintaining program; default '`ar`'.

`AS` Program for doing assembly; default '`as`'.

`CC` Program for compiling C programs; default '`cc`'.

`CXX` Program for compiling C++ programs; default '`g++`'.

`CO` Program for extracting a file from RCS; default '`co`'.

`CPP` Program for running the C preprocessor, with results to standard output; default '`$(CC) -E`'.

`FC` Program for compiling or preprocessing Fortran and Ratfor programs; default '`f77`'.

`GET` Program for extracting a file from SCCS; default '`get`'.

`LEX` Program to use to turn Lex grammars into C programs or Ratfor programs; default '`lex`'.

`PC` Program for compiling Pascal programs; default '`pc`'.

`YACC` Program to use to turn Yacc grammars into C programs; default '`yacc`'.

`YACCR` Program to use to turn Yacc grammars into Ratfor programs; default '`yacc -r`'.

`MAKEINFO` Program to convert a Texinfo source file into an Info file; default '`makeinfo`'.

`TEX` Program to make $\TeX$ DVI files from $\TeX$ source; default '`tex`'.

`TEXI2DVI` Program to make $\TeX$ DVI files from Texinfo source; default '`texi2dvi`'.

`WEAVE` Program to translate Web into $\TeX$; default '`weave`'.

`CWEAVE` Program to translate C Web into $\TeX$; default '`cweave`'.

`TANGLE` Program to translate Web into Pascal; default '`tangle`'.

`CTANGLE` Program to translate C Web into C; default '`ctangle`'.

`RM` Command to remove a file; default '`rm -f`'.

Here is a table of variables whose values are additional arguments for the programs above. The default values for all of these is the empty string, unless otherwise noted.

`ARFLAGS` Flags to give the archive-maintaining program; default '`rv`'.

`ASFLAGS` Extra flags to give to the assembler (when explicitly invoked on a '`.s`' or '`.S`' file).

`CFLAGS` Extra flags to give to the C compiler.

`CXXFLAGS` Extra flags to give to the C++ compiler.

`COFLAGS` Extra flags to give to the RCS `co` program.

`CPPFLAGS` Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers).

`FFLAGS` Extra flags to give to the Fortran compiler.

`GFLAGS` Extra flags to give to the SCCS `get` program.

`LDFLAGS` Extra flags to give to compilers when they are supposed to invoke the linker, '`ld`'.

`LFLAGS` Extra flags to give to Lex.

`PFLAGS` Extra flags to give to the Pascal compiler.

`RFLAGS` Extra flags to give to the Fortran compiler for Ratfor programs.

`YFLAGS` Extra flags to give to Yacc.

10.4 Chains of Implicit Rules

Sometimes a file can be made by a sequence of implicit rules. For example, a file '*n*.o' could be made from '*n*.y' by running first Yacc and then cc. Such a sequence is called a *chain*.

If the file '*n*.c' exists, or is mentioned in the makefile, no special searching is required: make finds that the object file can be made by C compilation from '*n*.c'; later on, when considering how to make '*n*.c', the rule for running Yacc is used. Ultimately both '*n*.c' and '*n*.o' are updated.

However, even if '*n*.c' does not exist and is not mentioned, make knows how to envision it as the missing link between '*n*.o' and '*n*.y'! In this case, '*n*.c' is called an *intermediate file*. Once make has decided to use the intermediate file, it is entered in the data base as if it had been mentioned in the makefile, along with the implicit rule that says how to create it.

Intermediate files are remade using their rules just like all other files. The difference is that the intermediate file is deleted when make is finished. Therefore, the intermediate file which did not exist before make also does not exist after make. The deletion is reported to you by printing a 'rm -f' command that shows what make is doing. (You can list the target pattern of an implicit rule (such as '%.o') as a dependency of the special target .PRECIOUS to preserve intermediate files made by implicit rules whose target patterns match that file's name; see Section 5.5 "Interrupts," page 41.)

A chain can involve more than two implicit rules. For example, it is possible to make a file 'foo' from 'RCS/foo.y,v' by running RCS, Yacc and cc. Then both 'foo.y' and 'foo.c' are intermediate files that are deleted at the end.

No single implicit rule can appear more than once in a chain. This means that make will not even consider such a ridiculous thing as making 'foo' from 'foo.o.o' by running the linker twice. This constraint has the added benefit of preventing any infinite loop in the search for an implicit rule chain.

There are some special implicit rules to optimize certain cases that would otherwise be handled by rule chains. For example, making 'foo' from 'foo.c' could be handled by compiling and linking with separate chained rules, using 'foo.o' as an intermediate file. But what actually happens is that a special rule for this case does the compilation and linking with a single cc command. The optimized rule is used in preference to the step-by-step chain because it comes earlier in the ordering of rules.

10.5 Defining and Redefining Pattern Rules

You define an implicit rule by writing a *pattern rule*. A pattern rule looks like an ordinary rule, except that its target contains the character '%' (exactly one of them). The target is considered a pattern for matching file names; the '%' can match any nonempty substring, while other characters match only themselves. The dependencies likewise use '%' to show how their names relate to the target name.

Thus, a pattern rule '%.o : %.c' says how to make any file '*stem*.o' from another file '*stem*.c'.

Note that expansion using '%' in pattern rules occurs **after** any variable or function expansions, which take place when the makefile is read. See Chapter 6 "How to Use Variables," page 51, and Chapter 8 "Functions for Transforming Text," page 71.

10.5.1 Introduction to Pattern Rules

A pattern rule contains the character '%' (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the '%' matches any nonempty substring, while other characters match only themselves.

For example, '%.c' as a pattern matches any file name that ends in '.c'. 's.%.c' as a pattern matches any file name that starts with 's.', ends in '.c' and is at least five characters long. (There must be at least one character to match the '%'.) The substring that the '%' matches is called the *stem*.

'%' in a dependency of a pattern rule stands for the same stem that was matched by the '%' in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration, and its dependency patterns must name files that exist or can be made. These files become dependencies of the target.

Thus, a rule of the form

```
%.o : %.c ; command...
```

specifies how to make a file '*n*.o', with another file '*n*.c' as its dependency, provided that '*n*.c' exists or can be made.

There may also be dependencies that do not use '%'; such a dependency attaches to every file made by this pattern rule. These unvarying dependencies are useful occasionally.

A pattern rule need not have any dependencies that contain '%', or in fact any dependencies at all. Such a rule is effectively a general wildcard. It provides a way to make any file that matches the target pattern. See Section 10.6 "Last Resort," page 111.

Pattern rules may have more than one target. Unlike normal rules, this does not act as many different rules with the same dependencies and commands. If a pattern rule has multiple targets, `make` knows that the rule's commands are responsible for making all of the targets. The commands are executed only once to make all the targets. When searching for a pattern rule to match a target, the target patterns of a rule other than the one that matches the target in need of a rule are incidental: `make` worries only about giving commands and dependencies to the file presently in question. However, when this file's commands are run, the other targets are marked as having been updated themselves.

The order in which pattern rules appear in the makefile is important since this is the order in which they are considered. Of equally applicable rules, only the first one found is used. The rules you write take precedence over those that are built in. Note however, that a rule whose dependencies actually exist or are mentioned always takes priority over a rule with dependencies that must be made by chaining other implicit rules.

10.5.2 Pattern Rule Examples

Here are some examples of pattern rules actually predefined in `make`. First, the rule that compiles '`.c`' files into '`.o`' files:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

defines a rule that can make any file '`x.o`' from '`x.c`'. The command uses the automatic variables '`$@`' and '`$<`' to substitute the names of the target file and the source file in each case where the rule applies (see Section 10.5.3 "Automatic Variables," page 106).

Here is a second built-in rule:

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

defines a rule that can make any file '`x`' whatsoever from a corresponding file '`x,v`' in the subdirectory '`RCS`'. Since the target is '`%`', this rule will apply to any file whatever, provided the appropriate dependency file exists. The double colon makes the rule *terminal*, which means that its dependency may not be an intermediate file (see Section 10.5.5 "Match-Anything Pattern Rules," page 109).

This pattern rule has two targets:

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

This tells `make` that the command '`bison -d x.y`' will make both '`x.tab.c`' and '`x.tab.h`'. If the file '`foo`' depends on the files '`parse.tab.o`' and

'scan.o' and the file 'scan.o' depends on the file 'parse.tab.h', when 'parse.y' is changed, the command 'bison -d parse.y' will be executed only once, and the dependencies of both 'parse.tab.o' and 'scan.o' will be satisfied. (Presumably the file 'parse.tab.o' will be recompiled from 'parse.tab.c' and the file 'scan.o' from 'scan.c', while 'foo' is linked from 'parse.tab.o', 'scan.o', and its other dependencies, and it will execute happily ever after.)

10.5.3 Automatic Variables

Suppose you are writing a pattern rule to compile a '.c' file into a '.o' file: how do you write the 'cc' command so that it operates on the right source file name? You cannot write the name in the command, because the name is different each time the implicit rule is applied.

What you do is use a special feature of make, the *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. In this example, you would use '$@' for the object file name and '$<' for the source file name.

Here is a table of automatic variables:

$@
: The file name of the target of the rule. If the target is an archive member, then '$@' is the name of the archive file. In a pattern rule that has multiple targets (see Section 10.5.1 "Introduction to Pattern Rules," page 104), '$@' is the name of whichever target caused the rule's commands to be run.

$%
: The target member name, when the target is an archive member. See Chapter 11 "Archives," page 115. For example, if the target is 'foo.a(bar.o)' then '$%' is 'bar.o' and '$@' is 'foo.a'. '$%' is empty when the target is not an archive member.

$<
: The name of the first dependency. If the target got its commands from an implicit rule, this will be the first dependency added by the implicit rule (see Chapter 10 "Implicit Rules," page 95).

$?
: The names of all the dependencies that are newer than the target, with spaces between them. For dependencies which are archive members, only the member named is used (see Chapter 11 "Archives," page 115).

$^
: The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used (see Chapter 11 "Archives," page 115).

A target has only one dependency on each other file it depends on, no matter how many times each file is listed as a dependency. So if you list a dependency more than once for a target, the value of `$^` contains just one copy of the name.

`$+`
: This is like '`$^`', but dependencies listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use in linking commands where it is meaningful to repeat library file names in a particular order.

`$*`
: The stem with which an implicit rule matches (see Section 10.5.4 "How Patterns Match," page 108). If the target is '`dir/a.foo.b`' and the target pattern is '`a.%.b`' then the stem is '`dir/foo`'. The stem is useful for constructing names of related files.

In a static pattern rule, the stem is part of the file name that matched the '`%`' in the target pattern.

In an explicit rule, there is no stem; so '`$*`' cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see Section 10.7 "Old-Fashioned Suffix Rules," page 111), '`$*`' is set to the target name minus the suffix. For example, if the target name is '`foo.c`', then '`$*`' is set to '`foo`', since '`.c`' is a suffix. GNU `make` does this bizarre thing only for compatibility with other implementations of `make`. You should generally avoid using '`$*`' except in implicit rules or static pattern rules.

If the target name in an explicit rule does not end with a recognized suffix, '`$*`' is set to the empty string for that rule.

'`$?`' is useful even in explicit rules when you wish to operate on only the dependencies that have changed. For example, suppose that an archive named '`lib`' is supposed to contain copies of several object files. This rule copies just the changed object files into the archive:

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

Of the variables listed above, four have values that are single file names, and two have values that are lists of file names. These six have variants that get just the file's directory name or just the file name within the directory. The variant variables' names are formed by appending '`D`' or '`F`', respectively. These variants are semi-obsolete in GNU `make` since the functions `dir` and `notdir` can be used to get a similar effect (see Section 8.3 "Functions for File Names," page 75). Note, however, that the '`F`' variants all omit the trailing slash which always appears in the output of the `dir` function. Here is a table of the variants:

'$(@D)'
: The directory part of the file name of the target, with the trailing slash removed. If the value of '$@' is 'dir/foo.o' then '$(@D)' is 'dir'. This value is '.' if '$@' does not contain a slash.

'$(@F)'
: The file-within-directory part of the file name of the target. If the value of '$@' is 'dir/foo.o' then '$(@F)' is 'foo.o'. '$(@F)' is equivalent to '$(notdir $@)'.

'$(*D)'
'$(*F)'
: The directory part and the file-within-directory part of the stem; 'dir' and 'foo' in this example.

'$(%D)'
'$(%F)'
: The directory part and the file-within-directory part of the target archive member name. This makes sense only for archive member targets of the form '*archive*(*member*)' and is useful only when *member* may contain a directory name. (See Section 11.1 "Archive Members as Targets," page 115.)

'$(<D)'
'$(<F)'
: The directory part and the file-within-directory part of the first dependency.

'$(^D)'
'$(^F)'
: Lists of the directory parts and the file-within-directory parts of all dependencies.

'$(?D)'
'$(?F)'
: Lists of the directory parts and the file-within-directory parts of all dependencies that are newer than the target.

Note that we use a special stylistic convention when we talk about these automatic variables; we write "the value of '$<'", rather than "the variable <" as we would write for ordinary variables such as objects and CFLAGS. We think this convention looks more natural in this special case. Please do not assume it has a deep significance; '$<' refers to the variable named < just as '$(CFLAGS)' refers to the variable named CFLAGS. You could just as well use '$(<)' in place of '$<'.

10.5.4 How Patterns Match

A target pattern is composed of a '%' between a prefix and a suffix, either or both of which may be empty. The pattern matches a file name only if the file name starts with the prefix and ends with the suffix, without overlap. The text between the prefix and the suffix is called the *stem*. Thus, when the pattern '%.o' matches the file name 'test.o', the stem is 'test'. The pattern rule dependencies are turned into actual

file names by substituting the stem for the character '%'. Thus, if in the same example one of the dependencies is written as '%.c', it expands to 'test.c'.

When the target pattern does not contain a slash (and it usually does not), directory names in the file names are removed from the file name before it is compared with the target prefix and suffix. After the comparison of the file name to the target pattern, the directory names, along with the slash that ends them, are added on to the dependency file names generated from the pattern rule's dependency patterns and the file name. The directories are ignored only for the purpose of finding an implicit rule to use, not in the application of that rule. Thus, 'e%t' matches the file name 'src/eat', with 'src/a' as the stem. When dependencies are turned into file names, the directories from the stem are added at the front, while the rest of the stem is substituted for the '%'. The stem 'src/a' with a dependency pattern 'c%r' gives the file name 'src/car'.

10.5.5 Match-Anything Pattern Rules

When a pattern rule's target is just '%', it matches any file name whatever. We call these rules *match-anything* rules. They are very useful, but it can take a lot of time for make to think about them, because it must consider every such rule for each file name listed either as a target or as a dependency.

Suppose the makefile mentions 'foo.c'. For this target, make would have to consider making it by linking an object file 'foo.c.o', or by C compilation-and-linking in one step from 'foo.c.c', or by Pascal compilation-and-linking from 'foo.c.p', and many other possibilities.

We know these possibilities are ridiculous since 'foo.c' is a C source file, not an executable. If make did consider these possibilities, it would ultimately reject them, because files such as 'foo.c.o' and 'foo.c.p' would not exist. But these possibilities are so numerous that make would run very slowly if it had to consider them.

To gain speed, we have put various constraints on the way make considers match-anything rules. There are two different constraints that can be applied, and each time you define a match-anything rule you must choose one or the other for that rule.

One choice is to mark the match-anything rule as *terminal* by defining it with a double colon. When a rule is terminal, it does not apply unless its dependencies actually exist. Dependencies that could be made with other implicit rules are not good enough. In other words, no further chaining is allowed beyond a terminal rule.

For example, the built-in implicit rules for extracting sources from RCS and SCCS files are terminal; as a result, if the file 'foo.c,v' does not exist, make will not even consider trying to make it as an intermediate file from 'foo.c,v.o' or from 'RCS/SCCS/s.foo.c,v'. RCS and SCCS files are generally ultimate source files, which should not be remade from any other files; therefore, make can save time by not looking for ways to remake them.

If you do not mark the match-anything rule as terminal, then it is nonterminal. A nonterminal match-anything rule cannot apply to a file name that indicates a specific type of data. A file name indicates a specific type of data if some non-match-anything implicit rule target matches it.

For example, the file name 'foo.c' matches the target for the pattern rule '%.c : %.y' (the rule to run Yacc). Regardless of whether this rule is actually applicable (which happens only if there is a file 'foo.y'), the fact that its target matches is enough to prevent consideration of any nonterminal match-anything rules for the file 'foo.c'. Thus, make will not even consider trying to make 'foo.c' as an executable file from 'foo.c.o', 'foo.c.c', 'foo.c.p', etc.

The motivation for this constraint is that nonterminal match-anything rules are used for making files containing specific types of data (such as executable files) and a file name with a recognized suffix indicates some other specific type of data (such as a C source file).

Special built-in dummy pattern rules are provided solely to recognize certain file names so that nonterminal match-anything rules will not be considered. These dummy rules have no dependencies and no commands, and they are ignored for all other purposes. For example, the built-in implicit rule

```
%.p :
```

exists to make sure that Pascal source files such as 'foo.p' match a specific target pattern and thereby prevent time from being wasted looking for 'foo.p.o' or 'foo.p.c'.

Dummy pattern rules such as the one for '%.p' are made for every suffix listed as valid for use in suffix rules (see Section 10.7 "Old-Fashioned Suffix Rules," page 111).

10.5.6 Canceling Implicit Rules

You can override a built-in implicit rule (or one you have defined yourself) by defining a new pattern rule with the same target and dependencies, but different commands. When the new rule is defined, the built-in one is replaced. The new rule's position in the sequence of implicit rules is determined by where you write the new rule.

You can cancel a built-in implicit rule by defining a pattern rule with the same target and dependencies, but no commands. For example, the following would cancel the rule that runs the assembler:

```
%.o : %.s
```

10.6 Defining Last-Resort Default Rules

You can define a last-resort implicit rule by writing a terminal match-anything pattern rule with no dependencies (see Section 10.5.5 "Match-Anything Rules," page 109). This is just like any other pattern rule; the only thing special about it is that it will match any target. So such a rule's commands are used for all targets and dependencies that have no commands of their own and for which no other implicit rule applies.

For example, when testing a makefile, you might not care if the source files contain real data, only that they exist. Then you might do this:

```
%::
        touch $@
```

to cause all the source files needed (as dependencies) to be created automatically.

You can instead define commands to be used for targets for which there are no rules at all, even ones which don't specify commands. You do this by writing a rule for the target `.DEFAULT`. Such a rule's commands are used for all dependencies which do not appear as targets in any explicit rule, and for which no implicit rule applies. Naturally, there is no `.DEFAULT` rule unless you write one.

If you use `.DEFAULT` with no commands or dependencies:

```
.DEFAULT:
```

the commands previously stored for `.DEFAULT` are cleared. Then `make` acts as if you had never defined `.DEFAULT` at all.

If you do not want a target to get the commands from a match-anything pattern rule or `.DEFAULT`, but you also do not want any commands to be run for the target, you can give it empty commands (see Section 5.8 "Defining Empty Commands," page 48).

You can use a last-resort rule to override part of another makefile. See Section 3.6 "Overriding Part of Another Makefile," page 16.

10.7 Old-Fashioned Suffix Rules

Suffix rules are the old-fashioned way of defining implicit rules for `make`. Suffix rules are obsolete because pattern rules are more general and clearer. They are supported in GNU `make` for compatibility with old makefiles. They come in two kinds: *double-suffix* and *single-suffix*.

A double-suffix rule is defined by a pair of suffixes: the target suffix and the source suffix. It matches any file whose name ends with the target suffix. The corresponding implicit dependency is made by replacing the target suffix with the source suffix in the file name. A two-suffix rule whose target and source suffixes are '.o' and '.c' is equivalent to the pattern rule '%.o : %.c'.

A single-suffix rule is defined by a single suffix, which is the source suffix. It matches any file name, and the corresponding implicit dependency name is made by appending the source suffix. A single-suffix rule whose source suffix is '.c' is equivalent to the pattern rule '% : %.c'.

Suffix rule definitions are recognized by comparing each rule's target against a defined list of known suffixes. When make sees a rule whose target is a known suffix, this rule is considered a single-suffix rule. When make sees a rule whose target is two known suffixes concatenated, this rule is taken as a double-suffix rule.

For example, '.c' and '.o' are both on the default list of known suffixes. Therefore, if you define a rule whose target is '.c.o', make takes it to be a double-suffix rule with source suffix '.c' and target suffix '.o'. Here is the old-fashioned way to define the rule for compiling a C source file:

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Suffix rules cannot have any dependencies of their own. If they have any, they are treated as normal files with funny names, not as suffix rules. Thus, the rule:

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

tells how to make the file '.c.o' from the dependency file 'foo.h', and is not at all like the pattern rule:

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

which tells how to make '.o' files from '.c' files, and makes all '.o' files using this pattern rule also depend on 'foo.h'.

Suffix rules with no commands are also meaningless. They do not remove previous rules as do pattern rules with no commands (see Section 10.5.6 "Canceling Implicit Rules," page 110). They simply enter the suffix or pair of suffixes concatenated as a target in the data base.

The known suffixes are simply the names of the dependencies of the special target .SUFFIXES. You can add your own suffixes by writing a rule for .SUFFIXES that adds more dependencies, as in:

```
.SUFFIXES: .hack .win
```

which adds '.hack' and '.win' to the end of the list of suffixes.

If you wish to eliminate the default known suffixes instead of just adding to them, write a rule for `.SUFFIXES` with no dependencies. By special dispensation, this eliminates all existing dependencies of `.SUFFIXES`. You can then write another rule to add the suffixes you want. For example,

```
.SUFFIXES:              # Delete the default suffixes
.SUFFIXES: .c .o .h     # Define our suffix list
```

The '`-r`' or '`--no-builtin-rules`' flag causes the default list of suffixes to be empty.

The variable `SUFFIXES` is defined to the default list of suffixes before `make` reads any makefiles. You can change the list of suffixes with a rule for the special target `.SUFFIXES`, but that does not alter this variable.

10.8 Implicit Rule Search Algorithm

Here is the procedure `make` uses for searching for an implicit rule for a target *t*. This procedure is followed for each double-colon rule with no commands, for each target of ordinary rules none of which have commands, and for each dependency that is not the target of any rule. It is also followed recursively for dependencies that come from implicit rules, in the search for a chain of rules.

Suffix rules are not mentioned in this algorithm because suffix rules are converted to equivalent pattern rules once the makefiles have been read in.

For an archive member target of the form '*archive*(*member*)', the following algorithm is run twice, first using the entire target name *t*, and second using '(*member*)' as the target *t* if the first run found no rule.

1. Split *t* into a directory part, called *d*, and the rest, called *n*. For example, if *t* is '`src/foo.o`', then *d* is '`src/`' and *n* is '`foo.o`'.
2. Make a list of all the pattern rules one of whose targets matches *t* or *n*. If the target pattern contains a slash, it is matched against *t*; otherwise, against *n*.
3. If any rule in that list is *not* a match-anything rule, then remove all nonterminal match-anything rules from the list.
4. Remove from the list all rules with no commands.
5. For each pattern rule in the list:
   a. Find the stem *s*, which is the nonempty part of *t* or *n* matched by the '`%`' in the target pattern.
   b. Compute the dependency names by substituting *s* for '`%`'; if the target pattern does not contain a slash, append *d* to the front of each dependency name.

   c. Test whether all the dependencies exist or ought to exist. (If a file name is mentioned in the makefile as a target or as an explicit dependency, then we say it ought to exist.)

      If all dependencies exist or ought to exist, or there are no dependencies, then this rule applies.

6. If no pattern rule has been found so far, try harder. For each pattern rule in the list:
   a. If the rule is terminal, ignore it and go on to the next rule.
   b. Compute the dependency names as before.
   c. Test whether all the dependencies exist or ought to exist.
   d. For each dependency that does not exist, follow this algorithm recursively to see if the dependency can be made by an implicit rule.
   e. If all dependencies exist, ought to exist, or can be made by implicit rules, then this rule applies.
7. If no implicit rule applies, the rule for `.DEFAULT`, if any, applies. In that case, give *t* the same commands that `.DEFAULT` has. Otherwise, there are no commands for *t*.

Once a rule that applies has been found, for each target pattern of the rule other than the one that matched *t* or *n*, the '`%`' in the pattern is replaced with *s* and the resultant file name is stored until the commands to remake the target file *t* are executed. After these commands are executed, each of these stored file names are entered into the data base and marked as having been updated and having the same update status as the file *t*.

When the commands of a pattern rule are executed for *t*, the automatic variables are set corresponding to the target and dependencies. See Section 10.5.3 "Automatic Variables," page 106.

11 Using `make` to Update Archive Files

Archive files are files containing named subfiles called *members*; they are maintained with the program `ar` and their main use is as subroutine libraries for linking.

11.1 Archive Members as Targets

An individual member of an archive file can be used as a target or dependency in `make`. You specify the member named *member* in archive file *archive* as follows:

```
archive(member)
```

This construct is available only in targets and dependencies, not in commands! Most programs that you might use in commands do not support this syntax and cannot act directly on archive members. Only `ar` and other programs specifically designed to operate on archives can do so. Therefore, valid commands to update an archive member target probably must use `ar`. For example, this rule says to create a member '`hack.o`' in archive '`foolib`' by copying the file '`hack.o`':

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

In fact, nearly all archive member targets are updated in just this way and there is an implicit rule to do it for you. **Note:** The '`c`' flag to `ar` is required if the archive file does not already exist.

To specify several members in the same archive, you can write all the member names together between the parentheses. For example:

```
foolib(hack.o kludge.o)
```

is equivalent to:

```
foolib(hack.o) foolib(kludge.o)
```

You can also use shell-style wildcards in an archive member reference. See Section 4.2 "Using Wildcard Characters in File Names," page 18. For example, '`foolib(*.o)`' expands to all existing members of the '`foolib`' archive whose names end in '`.o`'; perhaps '`foolib(hack.o) foolib(kludge.o)`'.

11.2 Implicit Rule for Archive Member Targets

Recall that a target that looks like '`a(m)`' stands for the member named *m* in the archive file *a*.

When `make` looks for an implicit rule for such a target, as a special feature it considers implicit rules that match '`(m)`', as well as those that match the actual target '`a(m)`'.

This causes one special rule whose target is '(%)' to match. This rule updates the target '*a*(*m*)' by copying the file *m* into the archive. For example, it will update the archive member target 'foo.a(bar.o)' by copying the *file* 'bar.o' into the archive 'foo.a' as a *member* named 'bar.o'.

When this rule is chained with others, the result is very powerful. Thus, 'make "foo.a(bar.o)"' (the quotes are needed to protect the '(' and ')' from being interpreted specially by the shell) in the presence of a file 'bar.c' is enough to cause the following commands to be run, even without a makefile:

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

Here make has envisioned the file 'bar.o' as an intermediate file. See Section 10.4 “Chains of Implicit Rules,” page 103.

Implicit rules such as this one are written using the automatic variable '$%'. See Section 10.5.3 “Automatic Variables,” page 106.

An archive member name in an archive cannot contain a directory name, but it may be useful in a makefile to pretend that it does. If you write an archive member target 'foo.a(dir/file.o)', make will perform automatic updating with this command:

```
ar r foo.a dir/file.o
```

which has the effect of copying the file 'dir/file.o' into a member named 'file.o'. In connection with such usage, the automatic variables %D and %F may be useful.

11.2.1 Updating Archive Symbol Directories

An archive file that is used as a library usually contains a special member named '__.SYMDEF' that contains a directory of the external symbol names defined by all the other members. After you update any other members, you need to update '__.SYMDEF' so that it will summarize the other members properly. This is done by running the ranlib program:

```
ranlib archivefile
```

Normally you would put this command in the rule for the archive file, and make all the members of the archive file dependencies of that rule. For example,

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

The effect of this is to update archive members 'x.o', 'y.o', etc., and then update the symbol directory member '__.SYMDEF' by running ranlib. The rules for updating the members are not shown here; most likely

you can omit them and use the implicit rule which copies files into the archive, as described in the preceding section.

This is not necessary when using the GNU `ar` program, which updates the '`__.SYMDEF`' member automatically.

11.3 Dangers When Using Archives

It is important to be careful when using parallel execution (the `-j` switch; see Section 5.3 "Parallel Execution," page 38) and archives. If multiple `ar` commands run at the same time on the same archive file, they will not know about each other and can corrupt the file.

Possibly a future version of `make` will provide a mechanism to circumvent this problem by serializing all commands that operate on the same archive file. But for the time being, you must either write your makefiles to avoid this problem in some other way, or not use `-j`.

11.4 Suffix Rules for Archive Files

You can write a special kind of suffix rule for dealing with archive files. See Section 10.7 "Suffix Rules," page 111, for a full explanation of suffix rules. Archive suffix rules are obsolete in GNU `make`, because pattern rules for archives are a more general mechanism (see Section 11.2 "Archive Update," page 115). But they are retained for compatibility with other `make`s.

To write a suffix rule for archives, you simply write a suffix rule using the target suffix '`.a`' (the usual suffix for archive files). For example, here is the old-fashioned suffix rule to update a library archive from C source files:

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

This works just as if you had written the pattern rule:

```
(%.o): %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

In fact, this is just what `make` does when it sees a suffix rule with '`.a`' as the target suffix. Any double-suffix rule '`.x.a`' is converted to a pattern rule with the target pattern '`(%.o)`' and a dependency pattern of '`%.x`'.

Since you might want to use '.a' as the suffix for some other kind of file, make also converts archive suffix rules to pattern rules in the normal way (see Section 10.7 "Suffix Rules," page 111). Thus a double-suffix rule '.*x*.a' produces two pattern rules: '(%.o): %.*x*' and '%.a: %.*x*'.

12 Features of GNU make

Here is a summary of the features of GNU make, for comparison with and credit to other versions of make. We consider the features of make in 4.2 BSD systems as a baseline. If you are concerned with writing portable makefiles, you should use only the features of make *not* listed here or in Chapter 13 "Missing," page 123.

Many features come from the version of make in System V.

- The VPATH variable and its special meaning. See Section 4.3 "Searching Directories for Dependencies," page 20. This feature exists in System V make, but is undocumented. It is documented in 4.3 BSD make (which says it mimics System V's VPATH feature).
- Included makefiles. See Section 3.3 "Including Other Makefiles," page 12. Allowing multiple files to be included with a single directive is a GNU extension.
- Variables are read from and communicated via the environment. See Section 6.9 "Variables from the Environment," page 62.
- Options passed through the variable MAKEFLAGS to recursive invocations of make. See Section 5.6.3 "Communicating Options to a Sub-make," page 45.
- The automatic variable $% is set to the member name in an archive reference. See Section 10.5.3 "Automatic Variables," page 106.
- The automatic variables $@, $*, $<, $%, and $? have corresponding forms like $(@F) and $(@D). We have generalized this to $^ as an obvious extension. See Section 10.5.3 "Automatic Variables," page 106.
- Substitution variable references. See Section 6.1 "Basics of Variable References," page 51.
- The command-line options '-b' and '-m', accepted and ignored. In System V make, these options actually do something.
- Execution of recursive commands to run make via the variable MAKE even if '-n', '-q' or '-t' is specified. See Section 5.6 "Recursive Use of make," page 41.
- Support for suffix '.a' in suffix rules. See Section 11.4 "Archive Suffix Rules," page 117. This feature is obsolete in GNU make, because the general feature of rule chaining (see Section 10.4 "Chains of Implicit Rules," page 103) allows one pattern rule for installing members in an archive (see Section 11.2 "Archive Update," page 115) to be sufficient.
- The arrangement of lines and backslash-newline combinations in commands is retained when the commands are printed, so they

appear as they do in the makefile, except for the stripping of initial whitespace.

The following features were inspired by various other versions of make. In some cases it is unclear exactly which versions inspired which others.

- Pattern rules using '%'. This has been implemented in several versions of make. We're not sure who invented it first, but it's been spread around a bit. See Section 10.5 "Defining and Redefining Pattern Rules," page 104.
- Rule chaining and implicit intermediate files. This was implemented by Stu Feldman in his version of make for AT&T Eighth Edition Research Unix, and later by Andrew Hume of AT&T Bell Labs in his mk program (where he terms it "transitive closure"). We do not really know if we got this from either of them or thought it up ourselves at the same time. See Section 10.4 "Chains of Implicit Rules," page 103.
- The automatic variable $^ containing a list of all dependencies of the current target. We did not invent this, but we have no idea who did. See Section 10.5.3 "Automatic Variables," page 106. The automatic variable $+ is a simple extension of $^.
- The "what if" flag ('-W' in GNU make) was (as far as we know) invented by Andrew Hume in mk. See Section 9.3 "Instead of Executing the Commands," page 85.
- The concept of doing several things at once (parallelism) exists in many incarnations of make and similar programs, though not in the System V or BSD implementations. See Section 5.2 "Command Execution," page 38.
- Modified variable references using pattern substitution come from SunOS 4. See Section 6.1 "Basics of Variable References," page 51. This functionality was provided in GNU make by the patsubst function before the alternate syntax was implemented for compatibility with SunOS 4. It is not altogether clear who inspired whom, since GNU make had patsubst before SunOS 4 was released.
- The special significance of '+' characters preceding command lines (see Section 9.3 "Instead of Executing the Commands," page 85) is mandated by *IEEE Standard 1003.2-1992* (POSIX.2).
- The '+=' syntax to append to the value of a variable comes from SunOS 4 make. See Section 6.6 "Appending More Text to Variables," page 59.
- The syntax '*archive*(*mem1 mem2*...)' to list multiple members in a single archive file comes from SunOS 4 make. See Section 11.1 "Archive Members," page 115.

- The -include directive to include makefiles with no error for a nonexistent file comes from SunOS 4 make. (But note that SunOS 4 make does not allow multiple makefiles to be specified in one -include directive.)

The remaining features are inventions new in GNU make:

- Use the '-v' or '--version' option to print version and copyright information.
- Use the '-h' or '--help' option to summarize the options to make.
- Simply-expanded variables. See Section 6.2 "The Two Flavors of Variables," page 52.
- Pass command-line variable assignments automatically through the variable MAKE to recursive make invocations. See Section 5.6 "Recursive Use of make," page 41.
- Use the '-C' or '--directory' command option to change directory. See Section 9.7 "Summary of Options," page 89.
- Make verbatim variable definitions with define. See Section 6.8 "Defining Variables Verbatim," page 61.
- Declare phony targets with the special target .PHONY.

  Andrew Hume of AT&T Bell Labs implemented a similar feature with a different syntax in his mk program. This seems to be a case of parallel discovery. See Section 4.4 "Phony Targets," page 24.
- Manipulate text by calling functions. See Chapter 8 "Functions for Transforming Text," page 71.
- Use the '-o' or '--old-file' option to pretend a file's modification-time is old. See Section 9.4 "Avoiding Recompilation of Some Files," page 87.
- Conditional execution.

  This feature has been implemented numerous times in various versions of make; it seems a natural extension derived from the features of the C preprocessor and similar macro languages and is not a revolutionary concept. See Chapter 7 "Conditional Parts of Makefiles," page 65.
- Specify a search path for included makefiles. See Section 3.3 "Including Other Makefiles," page 12.
- Specify extra makefiles to read with an environment variable. See Section 3.4 "The Variable MAKEFILES," page 14.
- Strip leading sequences of './' from file names, so that './*file*' and '*file*' are considered to be the same file.
- Use a special search method for library dependencies written in the form '-l*name*'. See Section 4.3.5 "Directory Search for Link Libraries," page 23.

- Allow suffixes for suffix rules (see Section 10.7 “Old-Fashioned Suffix Rules,” page 111) to contain any characters. In other versions of make, they must begin with ‘.’ and not contain any ‘/’ characters.
- Keep track of the current level of make recursion using the variable MAKELEVEL. See Section 5.6 “Recursive Use of make,” page 41.
- Specify static pattern rules. See Section 4.10 “Static Pattern Rules,” page 30.
- Provide selective vpath search. See Section 4.3 “Searching Directories for Dependencies,” page 20.
- Provide computed variable references. See Section 6.1 “Basics of Variable References,” page 51.
- Update makefiles. See Section 3.5 “How Makefiles Are Remade,” page 14. System V make has a very, very limited form of this functionality in that it will check out SCCS files for makefiles.
- Various new built-in implicit rules. See Section 10.2 “Catalogue of Implicit Rules,” page 96.
- The built-in variable ‘MAKE_VERSION’ gives the version number of make.

13 Incompatibilities and Missing Features

The `make` programs in various other systems support a few features that are not implemented in GNU `make`. The POSIX.2 standard (*IEEE Standard 1003.2-1992*) which specifies `make` does not require any of these features.

- A target of the form '`file((entry))`' stands for a member of archive file *file*. The member is chosen, not by name, but by being an object file which defines the linker symbol *entry*.

  This feature was not put into GNU `make` because of the nonmodularity of putting knowledge into `make` of the internal format of archive file symbol tables. See Section 11.2.1 "Updating Archive Symbol Directories," page 116.
- Suffixes (used in suffix rules) that end with the character '~' have a special meaning to System V `make`; they refer to the SCCS file that corresponds to the file one would get without the '~'. For example, the suffix rule '`.c~.o`' would make the file '`n.o`' from the SCCS file '`s.n.c`'. For complete coverage, a whole series of such suffix rules is required. See Section 10.7 "Old-Fashioned Suffix Rules," page 111.

  In GNU `make`, this entire series of cases is handled by two pattern rules for extraction from SCCS, in combination with the general feature of rule chaining. See Section 10.4 "Chains of Implicit Rules," page 103.
- In System V `make`, the string '`$$@`' has the strange meaning that, in the dependencies of a rule with multiple targets, it stands for the particular target that is being processed.

  This is not defined in GNU `make` because '`$$`' should always stand for an ordinary '`$`'.

  It is possible to get this functionality through the use of static pattern rules (see Section 4.10 "Static Pattern Rules," page 30). The System V `make` rule:

  ```
  $(targets): $$@.o lib.a
  ```

  can be replaced with the GNU `make` static pattern rule:

  ```
  $(targets): %: %.o lib.a
  ```
- In System V and 4.3 BSD `make`, files found by `VPATH` search (see Section 4.3 "Searching Directories for Dependencies," page 20) have their names changed inside command strings. We feel it is much cleaner to always use automatic variables and thus make this feature obsolete.
- In some Unix `make`s, the automatic variable `$*` appearing in the dependencies of a rule has the amazingly strange "feature" of expand-

ing to the full name of the *target of that rule*. We cannot imagine what went on in the minds of Unix make developers to do this; it is utterly inconsistent with the normal definition of $*.

- In some Unix makes, implicit rule search (see Chapter 10 “Using Implicit Rules,” page 95) is apparently done for *all* targets, not just those without commands. This means you can do:

  ```
  foo.o:
          cc -c foo.c
  ```

  and Unix make will intuit that ‘foo.o’ depends on ‘foo.c’.

  We feel that such usage is broken. The dependency properties of make are well-defined (for GNU make, at least), and doing such a thing simply does not fit the model.

- GNU make does not include any built-in implicit rules for compiling or preprocessing EFL programs. If we hear of anyone who is using EFL, we will gladly add them.
- It appears that in SVR4 make, a suffix rule can be specified with no commands, and it is treated as if it had empty commands (see Section 5.8 “Empty Commands,” page 48). For example:

  ```
  .c.a:
  ```

  will override the built-in ‘.c.a’ suffix rule.

  We feel that it is cleaner for a rule without commands to always simply add to the dependency list for the target. The above example can be easily rewritten to get the desired behavior in GNU make:

  ```
  .c.a: ;
  ```

- Some versions of make invoke the shell with the ‘-e’ flag, except under ‘-k’ (see Section 9.6 “Testing the Compilation of a Program,” page 88). The ‘-e’ flag tells the shell to exit as soon as any program it runs returns a nonzero status. We feel it is cleaner to write each shell command line to stand on its own and not require this special treatment.

14 Makefile Conventions

This chapter describes conventions for writing the Makefiles for GNU programs.

14.1 General Conventions for Makefiles

Every Makefile should contain this line:

```
SHELL = /bin/sh
```

to avoid trouble on systems where the SHELL variable might be inherited from the environment. (This is never a problem with GNU make.)

Different make programs have incompatible suffix lists and implicit rules, and this sometimes creates confusion or misbehavior. So it is a good idea to set the suffix list explicitly using only the suffixes you need in the particular Makefile, like this:

```
.SUFFIXES:
.SUFFIXES: .c .o
```

The first line clears out the suffix list, the second introduces all suffixes which may be subject to implicit rules in this Makefile.

Don't assume that '.' is in the path for command execution. When you need to run programs that are a part of your package during the make, please make sure that it uses './' if the program is built as part of the make or '$(srcdir)/' if the file is an unchanging part of the source code. Without one of these prefixes, the current search path is used.

The distinction between './' and '$(srcdir)/' is important when using the '--srcdir' option to 'configure'. A rule of the form:

```
foo.1 : foo.man sedscript
        sed -e sedscript foo.man > foo.1
```

will fail when the current directory is not the source directory, because 'foo.man' and 'sedscript' are not in the current directory.

When using GNU make, relying on 'VPATH' to find the source file will work in the case where there is a single dependency file, since the 'make' automatic variable '$<' will represent the source file wherever it is. (Many versions of make set '$<' only in implicit rules.) A makefile target like

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

should instead be written as

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

in order to allow 'VPATH' to work correctly. When the target has multiple dependencies, using an explicit '$(srcdir)' is the easiest way to make

the rule work well. For example, the target above for 'foo.1' is best written as:

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

14.2 Utilities in Makefiles

Write the Makefile commands (and any shell scripts, such as configure) to run in sh, not in csh. Don't use any special features of ksh or bash.

The configure script and the Makefile rules for building and installation should not use any utilities directly except these:

```
cat cmp cp echo egrep expr grep
ln mkdir mv pwd rm rmdir sed test touch
```

Stick to the generally supported options for these programs. For example, don't use 'mkdir -p', convenient as it may be, because most systems don't support it.

The Makefile rules for building and installation can also use compilers and related programs, but should do so via make variables so that the user can substitute alternatives. Here are some of the programs we mean:

```
ar bison cc flex install ld lex
make makeinfo ranlib texi2dvi yacc
```

Use the following make variables:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

When you use ranlib, you should make sure nothing bad happens if the system does not have ranlib. Arrange to ignore an error from that command, and print a message before the command to tell the user that failure of the ranlib command does not mean a problem.

If you use symbolic links, you should implement a fallback for systems that don't have symbolic links.

It is ok to use other utilities in Makefile portions (or scripts) intended only for particular systems where you know those utilities to exist.

14.3 Standard Targets for Users

All GNU programs should have the following targets in their Makefiles:

'all' Compile the entire program. This should be the default target. This target need not rebuild any documentation files;

Info files should normally be included in the distribution, and DVI files should be made only when explicitly asked for.

'install' Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use. If there is a simple test to verify that a program is properly installed, this target should run that test.

If possible, write the install target rule so that it does not modify anything in the directory where the program was built, provided 'make all' has just been done. This is convenient for building the program under one user name and installing it under another.

The commands should create all the directories in which files are to be installed, if they don't already exist. This includes the directories specified as the values of the variables prefix and exec_prefix, as well as all subdirectories that are needed. One way to do this is by means of an installdirs target as described below.

Use '-' before any command for installing a man page, so that make will ignore any errors. This is in case there are systems that don't have the Unix man page documentation system installed.

The way to install Info files is to copy them into '$(infodir)' with $(INSTALL_DATA) (see Section 14.4 "Command Variables," page 130), and then run the install-info program if it is present. install-info is a script that edits the Info 'dir' file to add or update the menu entry for the given Info file; it will be part of the Texinfo package. Here is a sample rule to install an Info file:

```
$(infodir)/foo.info: foo.info
# There may be a newer info file in . than in srcdir.
  -if test -f foo.info; then d=.; \
   else d=$(srcdir); fi; \
  $(INSTALL_DATA) $$d/foo.info $@; \
# Run install-info only if it exists.
# Use 'if' instead of just prepending '-' to the
# line so we notice real errors from install-info.
# We use '$(SHELL) -c' because some shells do not
# fail gracefully when there is an unknown command.
  if $(SHELL) -c 'install-info --version' \
     >/dev/null 2>&1; then \
    install-info --infodir=$(infodir) $$d/foo.info; \
  else true; fi
```

'uninstall'
: Delete all the installed files that the 'install' target would create (but not the noninstalled files such as 'make all' would create).

  This rule should not modify the directories where compilation is done, only the directories where files are installed.

'clean'
: Delete all files from the current directory that are normally created by building the program. Don't delete the files that record the configuration. Also preserve files that could be made by building, but normally aren't because the distribution comes with them.

  Delete '.dvi' files here if they are not part of the distribution.

'distclean'
: Delete all files from the current directory that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, 'make distclean' should leave only the files that were in the distribution.

'mostlyclean'
: Like 'clean', but may refrain from deleting a few files that people normally don't want to recompile. For example, the 'mostlyclean' target for GCC does not delete 'libgcc.a', because recompiling it is rarely necessary and takes a lot of time.

'maintainer-clean'
: Delete almost everything from the current directory that can be reconstructed with this Makefile. This typically includes everything deleted by distclean, plus more: C source files produced by Bison, tags tables, Info files, and so on.

  The reason we say "almost everything" is that 'make maintainer-clean' should not delete 'configure' even if 'configure' can be remade using a rule in the Makefile. More generally, 'make maintainer-clean' should not delete anything that needs to exist in order to run 'configure' and then begin to build the program. This is the only exception; maintainer-clean should delete everything else that can be rebuilt.

  The 'maintainer-clean' is intended to be used by a maintainer of the package, not by ordinary users. You may need special tools to reconstruct some of the files that 'make maintainer-clean' deletes. Since these files are normally

included in the distribution, we don't take care to make them easy to reconstruct. If you find you need to unpack the full distribution again, don't blame us.

To help make users aware of this, the commands for `maintainer-clean` should start with these two:

```
@echo "This command is intended for maintainers \
       to use;"
@echo "it deletes files that may require special \
       tools to rebuild."
```

'`TAGS`' Update a tags table for this program.

'`info`' Generate any Info files needed. The best way to write the rules is as follows:

```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
        $(MAKEINFO) $(srcdir)/foo.texi
```

You must define the variable `MAKEINFO` in the Makefile. It should run the `makeinfo` program, which is part of the Texinfo distribution.

'`dvi`' Generate DVI files for all TeXinfo documentation. For example:

```
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
```

You must define the variable `TEXI2DVI` in the Makefile. It should run the program `texi2dvi`, which is part of the Texinfo distribution. Alternatively, write just the dependencies, and allow GNU Make to provide the command.

'`dist`' Create a distribution tar file for this program. The tar file should be set up so that the file names in the tar file start with a subdirectory name which is the name of the package it is a distribution for. This name can include the version number.

For example, the distribution tar file of GCC version 1.40 unpacks into a subdirectory named '`gcc-1.40`'.

The easiest way to do this is to create a subdirectory appropriately named, use `ln` or `cp` to install the proper files in it, and then `tar` that subdirectory.

The `dist` target should explicitly depend on all non-source files that are in the distribution, to make sure they are up to date in the distribution. See section "Making Releases" in *GNU Coding Standards*.

'check' Perform self-tests (if any). The user must build the program before running the tests, but need not install the program; you should write the self-tests so that they work when the program is built but not installed.

The following targets are suggested as conventional names, for programs in which they are useful.

installcheck
: Perform installation tests (if any). The user must build and install the program before running the tests. You should not assume that '$(bindir)' is in the search path.

installdirs
: It's useful to add a target named 'installdirs' to create the directories where files are installed, and their parent directories. There is a script called 'mkinstalldirs' which is convenient for this; find it in the Texinfo package.You can use a rule like this:

```
# Make sure all installation directories
# (e.g. $(bindir)) actually exist by
# making them if necessary.
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

This rule should not modify the directories where compilation is done. It should do nothing but create installation directories.

14.4 Variables for Specifying Commands

Makefiles should provide variables for overriding certain commands, options, and so on.

In particular, you should run most utility programs via variables. Thus, if you use Bison, have a variable named BISON whose default value is set with 'BISON = bison', and refer to it with $(BISON) whenever you need to use Bison.

File management utilities such as ln, rm, mv, and so on, need not be referred to through variables in this way, since users don't need to replace them with other programs.

Each program-name variable should come with an options variable that is used to supply options to the program. Append 'FLAGS' to the program-name variable name to get the options variable name—for example, BISONFLAGS. (The name CFLAGS is an exception to this rule, but

we keep it because it is standard.) Use CPPFLAGS in any compilation command that runs the preprocessor, and use LDFLAGS in any compilation command that does linking as well as in any direct use of ld.

If there are C compiler options that *must* be used for proper compilation of certain files, do not include them in CFLAGS. Users expect to be able to specify CFLAGS freely themselves. Instead, arrange to pass the necessary options to the C compiler independently of CFLAGS, by writing them explicitly in the compilation commands or by defining an implicit rule, like this:

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

Do include the '-g' option in CFLAGS, because that is not *required* for proper compilation. You can consider it a default that is only recommended. If the package is set up so that it is compiled with GCC by default, then you might as well include '-O' in the default value of CFLAGS as well.

Put CFLAGS last in the compilation command, after other variables containing compiler options, so the user can use CFLAGS to override the others.

Every Makefile should define the variable INSTALL, which is the basic command for installing a file into the system.

Every Makefile should also define the variables INSTALL_PROGRAM and INSTALL_DATA. (The default for each of these should be $(INSTALL).) Then it should use those variables as the commands for actual installation, for executables and nonexecutables respectively. Use these variables as follows:

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

Always use a file name, not a directory name, as the second argument of the installation commands. Use a separate command for each file to be installed.

14.5 Variables for Installation Directories

Installation directories should always be named by variables, so it is easy to install in a nonstandard place. The standard names for these variables are described below. They are based on a standard filesystem layout; variants of it are used in SVR4, 4.4BSD, Linux, Ultrix v4, and other modern operating systems.

These two variables set the root for the installation. All the other installation directories should be subdirectories of one of these two, and nothing should be directly installed into these two directories.

'prefix'
: A prefix used in constructing the default values of the variables listed below. The default value of prefix should be '/usr/local' When building the complete GNU system, the prefix will be empty and '/usr' will be a symbolic link to '/'.

'exec_prefix'
: A prefix used in constructing the default values of some of the variables listed below. The default value of exec_prefix should be $(prefix).

  Generally, $(exec_prefix) is used for directories that contain machine-specific files (such as executables and subroutine libraries), while $(prefix) is used directly for other directories.

Executable programs are installed in one of the following directories.

'bindir'
: The directory for installing executable programs that users can run. This should normally be '/usr/local/bin', but write it as '$(exec_prefix)/bin'.

'sbindir'
: The directory for installing executable programs that can be run from the shell, but are only generally useful to system administrators. This should normally be '/usr/local/sbin', but write it as '$(exec_prefix)/sbin'.

'libexecdir'
: The directory for installing executable programs to be run by other programs rather than by users. This directory should normally be '/usr/local/libexec', but write it as '$(exec_prefix)/libexec'.

Data files used by the program during its execution are divided into categories in two ways.

- Some files are normally modified by programs; others are never normally modified (though users may edit some of these).
- Some files are architecture-independent and can be shared by all machines at a site; some are architecture-dependent and can be shared only by machines of the same kind and operating system; others may never be shared between two machines.

This makes for six different possibilities. However, we want to discourage the use of architecture-dependent files, aside from of object files and libraries. It is much cleaner to make other data files architecture-independent, and it is generally not hard.

Therefore, here are the variables makefiles should use to specify directories:

'datadir' The directory for installing read-only architecture independent data files. This should normally be '/usr/local/share', but write it as '$(prefix)/share'. As a special exception, see '$(infodir)' and '$(includedir)' below.

'sysconfdir'
The directory for installing read-only data files that pertain to a single machine–that is to say, files for configuring a host. Mailer and network configuration files, '/etc/passwd', and so forth belong here. All the files in this directory should be ordinary ASCII text files. This directory should normally be '/usr/local/etc', but write it as '$(prefix)/etc'.

Do not install executables in this directory (they probably belong in '$(libexecdir)' or '$(sbindir))'. Also do not install files that are modified in the normal course of their use (programs whose purpose is to change the configuration of the system excluded). Those probably belong in '$(localstatedir)'.

'sharedstatedir'
The directory for installing architecture-independent data files which the programs modify while they run. This should normally be '/usr/local/com', but write it as '$(prefix)/com'.

'localstatedir'
The directory for installing data files which the programs modify while they run, and that pertain to one specific machine. Users should never need to modify files in this directory to configure the package's operation; put such configuration information in separate files that go in 'datadir' or '$(sysconfdir)'. '$(localstatedir)' should normally be '/usr/local/var', but write it as '$(prefix)/var'.

'libdir' The directory for object files and libraries of object code. Do not install executables here, they probably belong in '$(libexecdir)' instead. The value of libdir should normally be '/usr/local/lib', but write it as '$(exec_prefix)/lib'.

'infodir' The directory for installing the Info files for this package. By default, it should be '/usr/local/info', but it should be written as '$(prefix)/info'.

'includedir'
: The directory for installing header files to be included by user programs with the C '#include' preprocessor directive. This should normally be '/usr/local/include', but write it as '$(prefix)/include'.

  Most compilers other than GCC do not look for header files in '/usr/local/include'. So installing the header files this way is only useful with GCC. Sometimes this is not a problem because some libraries are only really intended to work with GCC. But some libraries are intended to work with other compilers. They should install their header files in two places, one specified by includedir and one specified by oldincludedir.

'oldincludedir'
: The directory for installing '#include' header files for use with compilers other than GCC. This should normally be '/usr/include'.

  The Makefile commands should check whether the value of oldincludedir is empty. If it is, they should not try to use it; they should cancel the second installation of the header files.

  A package should not replace an existing header in this directory unless the header came from the same package. Thus, if your Foo package provides a header file 'foo.h', then it should install the header file in the oldincludedir directory if either (1) there is no 'foo.h' there or (2) the 'foo.h' that exists came from the Foo package.

  To tell whether 'foo.h' came from the Foo package, put a magic string in the file—part of a comment—and grep for that string.

Unix-style man pages are installed in one of the following:

'mandir'
: The directory for installing the man pages (if any) for this package. It should include the suffix for the proper section of the manual—usually '1' for a utility. It will normally be '/usr/local/man/man1', but you should write it as '$(prefix)/man/man1'.

'man1dir'
: The directory for installing section 1 man pages.

'man2dir'
: The directory for installing section 2 man pages.

'...'
: Use these names instead of 'mandir' if the package needs to install man pages in more than one section of the manual.

Don't make the primary documentation for any GNU software be a man page. Write a manual in Texinfo instead. Man pages are just for the sake of people running GNU software on Unix, which is a secondary application only.

'manext' The file name extension for the installed man page. This should contain a period followed by the appropriate digit; it should normally be '.1'.

'man1ext' The file name extension for installed section 1 man pages.

'man2ext' The file name extension for installed section 2 man pages.

'...' Use these names instead of 'manext' if the package needs to install man pages in more than one section of the manual.

And finally, you should set the following variable:

'srcdir' The directory for the sources being compiled. The value of this variable is normally inserted by the configure shell script.

For example:

```
# Common prefix for installation directories.
# NOTE: This directory must exist when you start the install.
prefix = /usr/local
exec_prefix = $(prefix)
# Where to put the executable for the command 'gcc'.
bindir = $(exec_prefix)/bin
# Where to put the directories used by the compiler.
libexecdir = $(exec_prefix)/libexec
# Where to put the Info files.
infodir = $(prefix)/info
```

If your program installs a large number of files into one of the standard user-specified directories, it might be useful to group them into a subdirectory particular to that program. If you do this, you should write the install rule to create these subdirectories.

Do not expect the user to include the subdirectory name in the value of any of the variables listed above. The idea of having a uniform set of variable names for installation directories is to enable the user to specify the exact same values for several different GNU packages. In order for this to be useful, all the packages must be designed so that they will work sensibly when the user does so.

Appendix A Quick Reference

This appendix summarizes the directives, text manipulation functions, and special variables which GNU `make` understands. See Section 4.7 “Special Targets,” page 27, Section 10.2 “Catalogue of Implicit Rules,” page 96, and Section 9.7 “Summary of Options,” page 89, for other summaries.

Here is a summary of the directives GNU `make` recognizes:

`define` *variable*
`endef`

: Define a multi-line, recursively-expanded variable.
See Section 5.7 “Sequences,” page 47.

`ifdef` *variable*
`ifndef` *variable*
`ifeq (`*a*`,`*b*`)`
`ifeq "`*a*`" "`*b*`"`
`ifeq '`*a*`' '`*b*`'`
`ifneq (`*a*`,`*b*`)`
`ifneq "`*a*`" "`*b*`"`
`ifneq '`*a*`' '`*b*`'`
`else`
`endif`

: Conditionally evaluate part of the makefile.
See Chapter 7 “Conditionals,” page 65.

`include` *file*

: Include another makefile.
See Section 3.3 “Including Other Makefiles,” page 12.

`override` *variable* `=` *value*
`override` *variable* `:=` *value*
`override` *variable* `+=` *value*
`override define` *variable*
`endef`

: Define a variable, overriding any previous definition, even one from the command line.
See Section 6.7 “The `override` Directive,” page 61.

`export`

: Tell `make` to export all variables to child processes by default.
See Section 5.6.2 “Communicating Variables to a Sub-`make`,” page 43.

`export` *variable*
`export` *variable* `=` *value*
`export` *variable* `:=` *value*
`export` *variable* `+=` *value*
`unexport` *variable*

Tell `make` whether or not to export a particular variable to child processes.
See Section 5.6.2 “Communicating Variables to a Sub-`make`,” page 43.

`vpath` *pattern path*

Specify a search path for files matching a ‘`%`’ pattern.
See Section 4.3.2 “The `vpath` Directive,” page 21.

`vpath` *pattern*

Remove all search paths previously specified for *pattern*.

`vpath`
Remove all search paths previously specified in any `vpath` directive.

Here is a summary of the text manipulation functions (see Chapter 8 “Functions,” page 71):

`$(subst` *from*`,`*to*`,`*text*`)`

Replace *from* with *to* in *text*.
See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(patsubst` *pattern*`,`*replacement*`,`*text*`)`

Replace words matching *pattern* with *replacement* in *text*.
See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(strip` *string*`)`

Remove excess whitespace characters from *string*.
See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(findstring` *find*`,`*text*`)`

Locate *find* in *text*.
See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(filter` *pattern*`...,`*text*`)`

Select words in *text* that match one of the *pattern* words.
See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(filter-out` *pattern*`...,`*text*`)`

Select words in *text* that *do not* match any of the *pattern* words.

See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(sort list)`
: Sort the words in *list* lexicographically, removing duplicates.
See Section 8.2 “Functions for String Substitution and Analysis,” page 72.

`$(dir names...)`
: Extract the directory part of each file name.
See Section 8.3 “Functions for File Names,” page 75.

`$(notdir names...)`
: Extract the non-directory part of each file name.
See Section 8.3 “Functions for File Names,” page 75.

`$(suffix names...)`
: Extract the suffix (the last ‘.’ and following characters) of each file name.
See Section 8.3 “Functions for File Names,” page 75.

`$(basename names...)`
: Extract the base name (name without suffix) of each file name.
See Section 8.3 “Functions for File Names,” page 75.

`$(addsuffix suffix,names...)`
: Append *suffix* to each word in *names*.
See Section 8.3 “Functions for File Names,” page 75.

`$(addprefix prefix,names...)`
: Prepend *prefix* to each word in *names*.
See Section 8.3 “Functions for File Names,” page 75.

`$(join list1,list2)`
: Join two parallel lists of words.
See Section 8.3 “Functions for File Names,” page 75.

`$(word n,text)`
: Extract the *n*th word (one-origin) of *text*.
See Section 8.3 “Functions for File Names,” page 75.

`$(words text)`
: Count the number of words in *text*.
See Section 8.3 “Functions for File Names,” page 75.

`$(firstword names...)`
: Extract the first word of *names*.
See Section 8.3 “Functions for File Names,” page 75.

`$(wildcard pattern...)`
: Find file names matching a shell file name pattern (*not* a '%' pattern).
See Section 4.2.3 "The Function `wildcard`," page 20.

`$(shell command)`
: Execute a shell command and return its output.
See Section 8.6 "The `shell` Function," page 80.

`$(origin variable)`
: Return a string describing how the `make` variable *variable* was defined.
See Section 8.5 "The `origin` Function," page 79.

`$(foreach var,words,text)`
: Evaluate *text* with *var* bound to each word in *words*, and concatenate the results.
See Section 8.4 "The `foreach` Function," page 78.

Here is a summary of the automatic variables. See Section 10.5.3 "Automatic Variables," page 106, for full information.

`$@`
: The file name of the target.

`$%`
: The target member name, when the target is an archive member.

`$<`
: The name of the first dependency.

`$?`
: The names of all the dependencies that are newer than the target, with spaces between them. For dependencies which are archive members, only the member named is used (see Chapter 11 "Archives," page 115).

`$^`
`$+`
: The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used (see Chapter 11 "Archives," page 115). The value of `$^` omits duplicate dependencies, while `$+` retains them and preserves their order.

`$*`
: The stem with which an implicit rule matches (see Section 10.5.4 "How Patterns Match," page 108).

`$(@D)`
`$(@F)`
: The directory part and the file-within-directory part of `$@`.

`$(*D)`
`$(*F)`
: The directory part and the file-within-directory part of `$*`.

`$(%D)`
`$(%F)`
: The directory part and the file-within-directory part of `$%`.

`$(<D)`
`$(<F)` The directory part and the file-within-directory part of `$<`.

`$(^D)`
`$(^F)` The directory part and the file-within-directory part of `$^`.

`$(+D)`
`$(+F)` The directory part and the file-within-directory part of `$+`.

`$(?D)`
`$(?F)` The directory part and the file-within-directory part of `$?`.

These variables are used specially by GNU `make`:

`MAKEFILES`
: Makefiles to be read on every invocation of `make`.
See Section 3.4 "The Variable `MAKEFILES`," page 14.

`VPATH`
: Directory search path for files not found in the current directory.
See Section 4.3.1 "`VPATH` Search Path for All Dependencies," page 21.

`SHELL`
: The name of the system default command interpreter, usually '`/bin/sh`'. You can set `SHELL` in the makefile to change the shell used to run commands. See Section 5.2 "Command Execution," page 38.

`MAKE`
: The name with which `make` was invoked. Using this variable in commands has special meaning. See Section 5.6.1 "How the `MAKE` Variable Works," page 42.

`MAKELEVEL`
: The number of levels of recursion (sub-`make`s).
See Section 5.6.2 "Variables/Recursion," page 43.

`MAKEFLAGS`
: The flags given to `make`. You can set this in the environment or a makefile to set flags.
See Section 5.6.3 "Communicating Options to a Sub-`make`," page 45.

`SUFFIXES`
: The default list of suffixes before `make` reads any makefiles.

Appendix B Complex Makefile Example

Here is the makefile for the GNU `tar` program. This is a moderately complex makefile.

Because it is the first target, the default goal is '`all`'. An interesting feature of this makefile is that '`testpad.h`' is a source file automatically created by the `testpad` program, itself compiled from '`testpad.c`'.

If you type '`make`' or '`make all`', then `make` creates the '`tar`' executable, the '`rmt`' daemon that provides remote tape access, and the '`tar.info`' Info file.

If you type '`make install`', then `make` not only creates '`tar`', '`rmt`', and '`tar.info`', but also installs them.

If you type '`make clean`', then `make` removes the '`.o`' files, and the '`tar`', '`rmt`', '`testpad`', '`testpad.h`', and '`core`' files.

If you type '`make distclean`', then `make` not only removes the same files as does '`make clean`' but also the '`TAGS`', '`Makefile`', and '`config.status`' files. (Although it is not evident, this makefile (and '`config.status`') is generated by the user with the `configure` program, which is provided in the `tar` distribution, but is not shown here.)

If you type '`make realclean`', then `make` removes the same files as does '`make distclean`' and also removes the Info files generated from '`tar.texinfo`'.

In addition, there are targets `shar` and `dist` that create distribution kits.

```
# Generated automatically from Makefile.in by configure.
# Un*x Makefile for GNU tar program.
# Copyright (C) 1991 Free Software Foundation, Inc.
# This program is free software; you can redistribute
# it and/or modify it under the terms of the GNU
# General Public License . . .
. . .
. . .
SHELL = /bin/sh

#### Start of system configuration section. ####

srcdir = .
```

```
# If you use gcc, you should either run the
# fixincludes script that comes with it or else use
# gcc with the -traditional option.  Otherwise ioctl
# calls will be compiled incorrectly on some systems.
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

# Things you might add to DEFS:
# -DSTDC_HEADERS        If you have ANSI C headers and
#                       libraries.
# -DPOSIX               If you have POSIX.1 headers and
#                       libraries.
# -DBSD42               If you have sys/dir.h (unless
#                       you use -DPOSIX), sys/file.h,
#                       and st_blocks in `struct stat'.
# -DUSG                 If you have System V/ANSI C
#                       string and memory functions
#                       and headers, sys/sysmacros.h,
#                       fcntl.h, getcwd, no valloc,
#                       and ndir.h (unless
#                       you use -DDIRENT).
# -DNO_MEMORY_H         If USG or STDC_HEADERS but do not
#                       include memory.h.
# -DDIRENT              If USG and you have dirent.h
#                       instead of ndir.h.
# -DSIGTYPE=int         If your signal handlers
#                       return int, not void.
# -DNO_MTIO             If you lack sys/mtio.h
#                       (magtape ioctls).
# -DNO_REMOTE           If you do not have a remote shell
#                       or rexec.
# -DUSE_REXEC           To use rexec for remote tape
#                       operations instead of
#                       forking rsh or remsh.
# -DVPRINTF_MISSING     If you lack vprintf function
#                       (but have _doprnt).
# -DDOPRNT_MISSING      If you lack _doprnt function.
#                       Also need to define
#                       -DVPRINTF_MISSING.
# -DFTIME_MISSING       If you lack ftime system call.
# -DSTRSTR_MISSING      If you lack strstr function.
# -DVALLOC_MISSING      If you lack valloc function.
# -DMKDIR_MISSING       If you lack mkdir and
#                       rmdir system calls.
# -DRENAME_MISSING      If you lack rename system call.
# -DFTRUNCATE_MISSING   If you lack ftruncate
#                       system call.
# -DV7                  On Version 7 Unix (not
#                       tested in a long time).
# -DEMUL_OPEN3          If you lack a 3-argument version
```

```
#                       of open, and want to emulate it
#                       with system calls you do have.
# -DNO_OPEN3            If you lack the 3-argument open
#                       and want to disable the tar -k
#                       option instead of emulating open.
# -DXENIX               If you have sys/inode.h
#                       and need it 94 to be included.

DEFS =  -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
        -DVPRINTF_MISSING -DBSD42
# Set this to rtapelib.o unless you defined NO_REMOTE,
# in which case make it empty.
RTAPELIB = rtapelib.o
LIBS =
DEF_AR_FILE = /dev/rmt8
DEFBLOCKING = 20

CDEBUG = -g
CFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \
        -DDEF_AR_FILE=\"$(DEF_AR_FILE)\" \
        -DDEFBLOCKING=$(DEFBLOCKING)
LDFLAGS = -g
prefix = /usr/local
# Prefix for each installed program,
# normally empty or ‘g’.
binprefix =

# The directory to install tar in.
bindir = $(prefix)/bin

# The directory to install the info files in.
infodir = $(prefix)/info
#### End of system configuration section. ####

SRC1 =  tar.c create.c extract.c buffer.c \
        getoldopt.c update.c gnu.c mangle.c
SRC2 =  version.c list.c names.c diffarch.c \
        port.c wildmat.c getopt.c
SRC3 =  getopt1.c regex.c getdate.y
SRCS =  $(SRC1) $(SRC2) $(SRC3)
OBJ1 =  tar.o create.o extract.o buffer.o \
        getoldopt.o update.o gnu.o mangle.o
OBJ2 =  version.o list.o names.o diffarch.o \
        port.o wildmat.o getopt.o
OBJ3 =  getopt1.o regex.o getdate.o $(RTAPELIB)
OBJS =  $(OBJ1) $(OBJ2) $(OBJ3)
```

```
AUX =   README COPYING ChangeLog Makefile.in  \
        makefile.pc configure configure.in \
        tar.texinfo tar.info* texinfo.tex \
        tar.h port.h open3.h getopt.h regex.h \
        rmt.h rmt.c rtapelib.c alloca.c \
        msd_dir.h msd_dir.c tcexparg.c \
        level-0 level-1 backup-specs testpad.c
all:    tar rmt tar.info

tar:    $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
rmt:    rmt.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.c
tar.info: tar.texinfo
        makeinfo tar.texinfo
install: all
        $(INSTALL) tar $(bindir)/$(binprefix)tar
        -test ! -f rmt || $(INSTALL) rmt /etc/rmt
        $(INSTALLDATA) $(srcdir)/tar.info* $(infodir)
$(OBJS): tar.h port.h testpad.h
regex.o buffer.o tar.o: regex.h
# getdate.y has 8 shift/reduce conflicts.
testpad.h: testpad
        ./testpad
testpad: testpad.o
        $(CC) -o $@ testpad.o
TAGS:   $(SRCS)
        etags $(SRCS)
clean:
        rm -f *.o tar rmt testpad testpad.h core
distclean: clean
        rm -f TAGS Makefile config.status
realclean: distclean
        rm -f tar.info*
shar: $(SRCS) $(AUX)
        shar $(SRCS) $(AUX) | compress \
          > tar-`sed -e '/version_string/!d' \
                     -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
                     -e q
                     version.c`.shar.Z
```

```
dist: $(SRCS) $(AUX)
        echo tar-`sed \
             -e '/version_string/!d' \
             -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
             -e q
             version.c` > .fname
        -rm -rf `cat .fname`
        mkdir `cat .fname`
        ln $(SRCS) $(AUX) `cat .fname`
        -rm -rf `cat .fname` .fname
        tar chZf `cat .fname`.tar.Z `cat .fname`
tar.zoo: $(SRCS) $(AUX)
        -rm -rf tmp.dir
        -mkdir tmp.dir
        -rm tar.zoo
        for X in $(SRCS) $(AUX) ; do \
            echo $$X ; \
            sed 's/$$/^M/' $$X \
            > tmp.dir/$$X ; done
        cd tmp.dir ; zoo aM ../tar.zoo *
        -rm -rf tmp.dir
```

Index of Concepts

#

(comments), in commands 37
(comments), in makefile............. 11
#include............................ 33

$

$, in function call..................... 71
$, in rules............................ 17
$, in variable name.................. 55
$, in variable reference............... 51

%

%, in pattern rules................... 104
%, quoting in patsubst.............. 72
%, quoting in static pattern............ 31
%, quoting in vpath.................. 22
%, quoting with \ (backslash) .. 22, 31, 72

*

* (wildcard character)................. 18

,

,v (RCS file extension)................ 99

-

- (in commands)...................... 40
-, and define......................... 48
--assume-new 86, 93
--assume-new, and recursion........ 45
--assume-old 87, 91
--assume-old, and recursion........ 45
--debug............................. 89
--directory.................... 42, 89
--directory, and
  --print-directory........... 47
--directory, and recursion......... 45
--dry-run................... 37, 85, 91
--environment-overrides......... 90
--file....................... 12, 83, 90
--file, and recursion................ 45
--help.............................. 90
--ignore-errors 40, 90
--include-dir................. 13, 90
--jobs........................... 38, 90
--jobs, and recursion............... 45
--just-print............... 37, 85, 91
--keep-going............... 40, 88, 90
--load-average................ 39, 90
--makefile 12, 83, 90
--max-load..................... 39, 90
--new-file..................... 86, 93
--new-file, and recursion........... 45
--no-builtin-rules.............. 91
--no-keep-going.................. 92
--no-print-directory......... 47, 92
--old-file..................... 87, 91
--old-file, and recursion........... 45
--print-data-base................ 91
--print-directory................ 92
--print-directory, and
  --directory.................. 47
--print-directory, and recursion
  47
--print-directory, disabling...... 47
--question..................... 85, 91
--quiet......................... 37, 92
--recon 37, 85, 91
--silent........................ 37, 92
--stop.............................. 92
--touch......................... 85, 92
--touch, and recursion.............. 42
--version.......................... 92
--warn-undefined-variables..... 93
--what-if...................... 86, 93
-b................................... 89
-C.............................. 42, 89
-C, and -w........................... 47
-C, and recursion.................... 45
-d................................... 89
-e................................... 90
-e (shell flag)........................ 34
-f........................... 12, 83, 90
-f, and recursion.................... 45
-h................................... 90

-i 40, 90
-I 13, 90
-j 38, 90
-j, and archive update 117
-j, and recursion 45
-k 40, 88, 90
-l 90
-l (library search) 23
-l (load average) 39
-m 89
-M (to compiler) 33
-MM (to GNU compiler) 34
-n 37, 85, 91
-o 87, 91
-o, and recursion 45
-p 91
-q 85, 91
-r 91
-s 37, 92
-S 92
-t 85, 92
-t, and recursion 42
-v 92
-w 92
-W 86, 93
-w, and -C 47
-w, and recursion 47
-W, and recursion 45
-w, disabling 47

.

.a (archives) 117
.c 97
.C 97
.cc 97
.ch 99
.d 34
.def 97
.dvi 99
.f 97
.F 97
.info 99
.l 98
.ln 99
.mod 97
.o 97, 98
.p 97
.PRECIOUS intermediate files 103
.r 97
.s 98
.S 98
.sh 100
.sym 97
.tex 99
.texi 99
.texinfo 99
.txinfo 99
.w 99
.web 99
.y 98

:

:: rules (double-colon) 32
:= 53, 58

=

= 52, 58

?

? (wildcard character) 18

[

[...] (wildcard characters) 18

_

__.SYMDEF 116

@

@ (in commands) 37
@, and define 48

~

~ (tilde) 18

+

+, and define 48
+= 59

\

\ (backslash), for continuation lines 4
\ (backslash), in commands 38
\ (backslash), to quote % 22, 31, 72

A

all (standard target) 84
appending to variables 59
ar 101
archive 115
archive member targets 115
archive symbol directory updating . . . 116
archive, and -j 117
archive, and parallel execution 117
archive, suffix rule for 117
Arg list too long 46
arguments of functions 71
as 98, 101
assembly, rule to compile 98
automatic generation of dependencies 13, 33
automatic variables 106

B

backquotes 80
backslash (\), for continuation lines 4
backslash (\), in commands 38
backslash (\), to quote % 22, 31, 72
basename 76
broken pipe 39
bugs, reporting 2
built-in special targets 27

C

C, rule to compile 97
C++, rule to compile 97
cc 97, 101
cd (shell command) 38, 42
chains of rules 103
check (standard target) 85
clean (standard target) 84
clean target 5, 9
cleaning up 9
clobber (standard target) 84
co 99, 101
combining rules by dependency 8
command line variable definitions, and recursion 45
command line variables 87
commands 17
commands, backslash (\) in 38
commands, comments in 37
commands, echoing 37
commands, empty 48
commands, errors in 40
commands, execution 38
commands, execution in parallel 38
commands, expansion 80
commands, how to write 37
commands, instead of executing 85
commands, introduction to 3
commands, quoting newlines in 38
commands, sequences of 47
comments, in commands 37
comments, in makefile 11
compatibility 119
compatibility in exporting 44
compilation, testing 88
computed variable name 55
conditionals 65
continuation lines 4
conventions for makefiles 125
ctangle 99, 102
cweave 99, 102

D

deducing commands (implicit rules) 7
default goal 5, 17
default makefile name 12
default rules, last-resort 111
defining variables verbatim 61
deletion of target files 41
dependencies 18
dependencies, automatic generation of 13, 33
dependencies, introduction to 3
dependencies, list of all 107
dependencies, list of changed 106
dependencies, varying (static pattern) 30
dependency 17
dependency pattern, implicit 104
dependency pattern, static (not implicit) 31
directive 11
directories, printing them 47
directories, updating archive symbol 116
directory part 75

directory search (VPATH) 20
directory search (VPATH), and implicit rules 23
directory search (VPATH), and link libraries 23
directory search (VPATH), and shell commands 23
dist (standard target)................ 85
distclean (standard target) 84
dollar sign ($), in function call 71
dollar sign ($), in rules 17
dollar sign ($), in variable name 55
dollar sign ($), in variable reference .. 51
double-colon rules..................... 32
duplicate words, removing 74

E

E2BIG 46
echoing of commands 37
editor 3
Emacs (M-x compile) 41
empty commands 48
empty targets......................... 26
environment 62
environment, and recursion........... 43
environment, SHELL in 38
errors (in commands) 40
errors with wildcards 19
execution, in parallel.................. 38
execution, instead of 85
execution, of commands............... 38
exit status (errors) 40
explicit rule, definition of 11
exporting variables 43

F

f77 .... 97, 101
features of GNU make 119
features, missing 123
file name functions.................... 75
file name of makefile.................. 12
file name of makefile, how to specify .. 12
file name prefix, adding............... 76
file name suffix 76
file name suffix, adding 76
file name with wildcards.............. 18
file name, basename of................ 76
file name, directory part 75
file name, nondirectory part 75
files, assuming new 86
files, assuming old 87
files, avoiding recompilation of........ 87
files, intermediate 103
filtering out words 74
filtering words 74
finding strings 73
flags 89
flags for compilers 100
flavors of variables.................... 52
FORCE................................. 26
force targets 26
Fortran, rule to compile............... 97
functions.............................. 71
functions, for file names............... 75
functions, for text 72
functions, syntax of 71

G

g++ 97, 101
gcc 97
generating dependencies automatically 13, 33
get 99, 101
globbing (wildcards) 18
goal 5
goal, default 5, 17
goal, how to specify 83

H

home directory........................ 18

I

IEEE Standard 1003.2 1
implicit rule........................... 95
implicit rule, and directory search 23
implicit rule, and VPATH 23
implicit rule, definition of............. 11
implicit rule, how to use 95
implicit rule, introduction to 7
implicit rule, predefined 96
implicit rule, search algorithm....... 113
including (MAKEFILES variable) 14
including other makefiles 12
incompatibilities..................... 123

Info, rule to format 99
install (standard target) 85
intermediate files 103
intermediate files, preserving 103
interrupt 41

J

job slots 38
job slots, and recursion 45
jobs, limiting based on load 39
joining lists of words 76

K

killing (interruption) 41

L

last-resort default rules 111
ld 98
lex 98, 101
Lex, rule to run 98
libraries for linking, directory search 23
library archive, suffix rule for 117
limiting jobs based on load 39
link libraries, and directory search 23
linking, predefined rule for 98
lint 99
lint, rule to run 99
list of all dependencies 107
list of changed dependencies 106
load average 39
loops in variable expansion 53
lpr (shell command) 19, 26

M

m2c 97
macro 51
make depend 33
makefile 3
makefile name 12
makefile name, how to specify 12
makefile rule parts 3
makefile, and MAKEFILES variable 14
makefile, conventions for 125
makefile, how make processes 5
makefile, how to write 11
makefile, including 12
makefile, overriding 16
makefile, remaking of 14
makefile, simple 4
makeinfo 99, 101
match-anything rule 109
match-anything rule, used to override 16
missing features 123
mistakes with wildcards 19
modified variable reference 55
Modula-2, rule to compile 97
mostlyclean (standard target) 84
multiple rules for one target 29
multiple rules for one target (::) 32
multiple targets 28
multiple targets, in pattern rule 105

N

name of makefile 12
name of makefile, how to specify 12
nested variable reference 55
newline, quoting, in commands 38
newline, quoting, in makefile 4
nondirectory part 75

O

obj 6
OBJ 6
objects 6
OBJECTS 6
objs 6
OBJS 6
old-fashioned suffix rules 111
options 89
options, and recursion 45
options, setting from environment 46
options, setting in makefiles 46
order of pattern rules 105
origin of variable 79
overriding makefiles 16
overriding variables with arguments .. 87
overriding with override 61

P

parallel execution 38
parallel execution, and archive update 117

parts of makefile rule . . . 3
Pascal, rule to compile . . . 97
pattern rule . . . 104
pattern rules, order of . . . 105
pattern rules, static (not implicit) . . . 30
pattern rules, static, syntax of . . . 30
pc . . . 97, 101
phony targets . . . 24
pitfalls of wildcards . . . 19
portability . . . 119
POSIX . . . 1
POSIX.2 . . . 46
precious targets . . . 27
prefix, adding . . . 76
preserving intermediate files . . . 103
preserving with .PRECIOUS . . . 27, 103
print (standard target) . . . 85
print target . . . 19, 26
printing directories . . . 47
printing of commands . . . 37
problems and bugs, reporting . . . 2
problems with wildcards . . . 19
processing a makefile . . . 5

Q

question mode . . . 85
quoting %, in patsubst . . . 72
quoting %, in static pattern . . . 31
quoting %, in vpath . . . 22
quoting newline, in commands . . . 38
quoting newline, in makefile . . . 4

R

Ratfor, rule to compile . . . 97
RCS, rule to extract from . . . 99
README . . . 12
realclean (standard target) . . . 84
recompilation . . . 3
recompilation, avoiding . . . 87
recording events with empty targets . . . 26
recursion . . . 41
recursion, and -C . . . 45
recursion, and -f . . . 45
recursion, and -j . . . 45
recursion, and -o . . . 45
recursion, and -t . . . 42
recursion, and -w . . . 47
recursion, and -W . . . 45
recursion, and command line variable definitions . . . 45
recursion, and environment . . . 43
recursion, and MAKE variable . . . 42
recursion, and MAKEFILES variable . . . 14
recursion, and options . . . 45
recursion, and printing directories . . . 47
recursion, and variables . . . 43
recursion, level of . . . 44
recursive variable expansion . . . 51, 52
recursively expanded variables . . . 52
reference to variables . . . 51, 54
relinking . . . 6
remaking makefiles . . . 14
removal of target files . . . 41
removing duplicate words . . . 74
removing, to clean up . . . 9
reporting bugs . . . 2
rm . . . 102
rm (shell command) . . . 5, 18, 24, 40
rule commands . . . 37
rule dependencies . . . 18
rule syntax . . . 17
rule targets . . . 17
rule, and $ . . . 17
rule, double-colon (::) . . . 32
rule, explicit, definition of . . . 11
rule, how to write . . . 17
rule, implicit . . . 95
rule, implicit, and directory search . . . 23
rule, implicit, and VPATH . . . 23
rule, implicit, chains of . . . 103
rule, implicit, definition of . . . 11
rule, implicit, how to use . . . 95
rule, implicit, introduction to . . . 7
rule, implicit, predefined . . . 96
rule, introduction to . . . 3
rule, multiple for one target . . . 29
rule, no commands or dependencies . . . 26
rule, pattern . . . 104
rule, static pattern . . . 30
rule, static pattern versus implicit . . . 32
rule, with multiple targets . . . 28

S

s. (SCCS file prefix) . . . 99
SCCS, rule to extract from . . . 99

search algorithm, implicit rule 113
search path for dependencies (VPATH) 20
search path for dependencies (VPATH), and implicit rules 23
search path for dependencies (VPATH), and link libraries 23
searching for strings 73
sed (shell command) 34
selecting words 77
sequences of commands 47
setting options from environment 46
setting options in makefiles 46
setting variables 58
several rules for one target 29
several targets in a rule 28
shar (standard target) 85
shell command 5
shell command, and directory search . . 23
shell command, execution 38
shell command, function for 80
shell file name pattern (in include) . . 12
shell wildcards (in include) 12
signal 41
silent operation 37
simple makefile 4
simple variable expansion 51
simplifying with variables 6
simply expanded variables 53
sorting words 74
spaces, in variable values 54
spaces, stripping 73
special targets 27
specifying makefile name 12
standard input 39
standards conformance 1
standards for makefiles 125
static pattern rule 30
static pattern rule, syntax of 30
static pattern rule, versus implicit 32
stem 30, 108
stem, variable for 107
strings, searching for 73
stripping whitespace 73
sub-make 43
subdirectories, recursion for 41
substitution variable reference 55
suffix rule 111
suffix rule, for archive 117
suffix, adding 76
suffix, function to find 76
suffix, substituting in variables 55
switches 89
symbol directories, updating archive 116
syntax of rules 17

T

tab character (in commands) 17
tabs in rules 3
TAGS (standard target) 85
tangle 99, 102
tar (standard target) 85
target 17
target pattern, implicit 104
target pattern, static (not implicit) 30
target, deleting on error 41
target, deleting on interrupt 41
target, multiple in pattern rule 105
target, multiple rules for one 29
target, touching 85
targets 17
targets without a file 24
targets, built-in special 27
targets, empty 26
targets, force 26
targets, introduction to 3
targets, multiple 28
targets, phony 24
terminal rule 109
test (standard target) 85
testing compilation 88
tex 99, 102
TeX, rule to run 99
texi2dvi 99, 102
Texinfo, rule to format 99
tilde (~) 18
touch (shell command) 19, 26
touching files 85

U

undefined variables, warning message 93
updating archive symbol directories 116

updating makefiles 14

V

value 51
value, how a variable gets it 58
variable 51
variable definition 11
variables 6
variables, '$' in name 55
variables, and implicit rule 106
variables, appending to 59
variables, automatic 106
variables, command line 87
variables, command line, and recursion 45
variables, computed names 55
variables, defining verbatim 61
variables, environment 43, 62
variables, exporting 43
variables, flavors 52
variables, how they get their values... 58
variables, how to reference 51
variables, loops in expansion 53
variables, modified reference 55
variables, nested references 55
variables, origin of 79
variables, overriding 61
variables, overriding with arguments 87
variables, recursively expanded 52
variables, setting 58
variables, simply expanded 53
variables, spaces in values 54
variables, substituting suffix in 55
variables, substitution reference 55
variables, warning for undefined 93
varying dependencies 30
verbatim variable definition 61
vpath 20
VPATH, and implicit rules 23
VPATH, and link libraries 23

W

weave 99, 102
Web, rule to run 99
what if 86
whitespace, in variable values 54
whitespace, stripping 73
wildcard 18
wildcard pitfalls 19
wildcard, function 77
wildcard, in archive member 115
wildcard, in include 12
words, extracting first 77
words, filtering 74
words, filtering out 74
words, finding number 77
words, iterating over 78
words, joining lists 76
words, removing duplicates 74
words, selecting 77
writing rule commands 37
writing rules 17

Y

yacc 47, 98, 101
Yacc, rule to run 98

Index of Functions, Variables, & Directives

$

$% 106
$(%D) 108
$(%F) 108
$(*D) 108
$(*F) 108
$(?D) 108
$(?F) 108
$(@D) 108
$(@F) 108
$(^D) 108
$(^F) 108
$(<D) 108
$(<F) 108
$* 107
$*, and static pattern 32
$? 106
$@ 106
$+ 107
$^ 106
$< 106

%

% (automatic variable) 106
%D (automatic variable) 108
%F (automatic variable) 108

*

* (automatic variable) 107
* (automatic variable), unsupported
    bizarre usage 124
*D (automatic variable) 108
*F (automatic variable) 108

.

.DEFAULT 27, 111
.DEFAULT, and empty commands 48
.DELETE_ON_ERROR 41
.EXPORT_ALL_VARIABLES 28, 44
.IGNORE 27, 40
.PHONY 24, 27
.POSIX 46
.PRECIOUS 27, 41
.SILENT 28, 37
.SUFFIXES 27, 112

/

/usr/gnu/include 13
/usr/include 13
/usr/local/include 13

?

? (automatic variable) 106
?D (automatic variable) 108
?F (automatic variable) 108

@

@ (automatic variable) 106
@D (automatic variable) 108
@F (automatic variable) 108

+

+ (automatic variable) 107

^

^ (automatic variable) 106
^D (automatic variable) 108
^F (automatic variable) 108

<

< (automatic variable) 106
<D (automatic variable) 108
<F (automatic variable) 108

A

addprefix 76
addsuffix 76
AR 101
ARFLAGS 102
AS 101
ASFLAGS 102

B

basename............................ 76

C

CC.................................. 101
CFLAGS.............................. 102
CO.................................. 101
COFLAGS............................. 102
CPP................................. 101
CPPFLAGS............................ 102
CTANGLE............................. 102
CWEAVE.............................. 102
CXX................................. 101
CXXFLAGS............................ 102

D

define.............................. 61
dir................................. 75

E

else................................ 66
endef............................... 61
endif............................... 66
export.............................. 43

F

FC.................................. 101
FFLAGS.............................. 102
filter.............................. 74
filter-out.......................... 74
findstring.......................... 73
firstword........................... 77
foreach............................. 78

G

GET................................. 101
GFLAGS.............................. 102
GNUmakefile......................... 12

I

ifdef............................... 66
ifeq................................ 66
ifndef.............................. 66
ifneq............................... 66
include............................. 12

J

join................................ 76

L

LDFLAGS............................. 102
LEX................................. 101
LFLAGS.............................. 102

M

MAKE............................. 42, 53
makefile............................ 12
Makefile............................ 12
MAKEFILES........................ 14, 45
MAKEFLAGS........................... 45
MAKEINFO............................ 101
MAKELEVEL........................ 44, 53
MAKEOVERRIDES....................... 45
MFLAGS.............................. 46

N

notdir.............................. 75

O

origin.............................. 79
OUTPUT_OPTION....................... 100
override............................ 61

P

patsubst......................... 55, 72
PC.................................. 101
PFLAGS.............................. 102

R

RFLAGS.............................. 102
RM.................................. 102

S

shell............................... 80
SHELL............................... 38
SHELL (command execution).......... 38
sort................................ 74
strip............................... 73
subst............................ 29, 72
suffix.............................. 76
SUFFIXES............................ 113

T

TANGLE 102
TEX 101
TEXI2DVI 102

U

unexport 43

V

vpath 20, 21
VPATH 20, 21

W

WEAVE 102
wildcard 20, 77
word 77
words 77

Y

YACC 101
YACCR 101
YFLAGS 102

Developing With DOS

Development with the Cygnus Developer's Kit
In a DOS-hosted environment
September 1994

Cygnus Support

Copyright © 1994, 1995 Cygnus Support

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

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

Table of Contents

MS-DOS Host Notes for the Cygnus Developer's Kit **1**

1 Running the Programs **1**
- 1.1 Environment variables 1
- 1.2 The SHARE program 2
- 1.3 Typical initialization files 3
- 1.4 GO32, the 32-bit launcher 4
- 1.5 Remote debugging using asynctsr 4
- 1.6 Bug report form 5

2 Warnings for DOS Developer's Kits **5**
- 2.1 Memory requirements for MS-DOS 5
- 2.2 C++ compiler 6
- 2.3 Disk cache 6
- 2.4 Expanding environment space 6
- 2.5 Debugging AMD 29K programs 6
- 2.6 DEL does not work in Info 7
- 2.7 Includes with conflicting filenames 7

3 Hints and Common Problems **8**

4 Technical info about the release **9**

MS-DOS Host Notes for the Cygnus Developer's Kit

This note is a summary of the Cygnus Developer's Kit features specific to developing code with DOS. The DOS environment has a number of idiosyncrasies: we have attempted to share our expertise in development in this environment.

If you discover other useful information for inclusion in this note, please forward it to progressive@cygnus.com.

Installation procedures and specific warnings for this release are covered in the Installation notes.

1 Running the Programs

Before you run the cross compiler, you must set some environment variables and you must run the MS-DOS program 'SHARE.EXE'.

1.1 Environment variables

To run the compiler you first have to set some environment variables so that the tools can find each other. The environment variables need to refer to the directory where you installed your Developer's Kit.

A script which does this job, called 'SETENV.BAT', is automatically created when the install program is run. 'SETENV.BAT' is placed at the top level of the install directory.

These variables need to be reset each time your current session ends. You may wish to edit your 'AUTOEXEC.BAT' file to include the contents of 'SETENV.BAT', or you may want to run the script from the 'AUTOEXEC.BAT' file.

To set up the enviroment variables automatically when your session begins, add the following line to your 'AUTOEXEC.BAT' file (assuming the tools were installed into 'C:\CYGNUS')

```
CALL C:\CYGNUS\SETENV.BAT
```

Or you can put this into your 'AUTOEXEC.BAT':

```
set LIBRARY_PATH=C:\CYGNUS\LIB
set C_INCLUDE_PATH=C:\CYGNUS\INCLUDE
set INFOPATH=C:\CYGNUS\INFO
set GO32=EMU C:\CYGNUS\BIN\EMU387
```

1.2 The SHARE program

To use the cross-development tools in your Developer's Kit, you must first load the MS-DOS system program SHARE.EXE. Doing this in 'AUTOEXEC.BAT' is generally a good idea. Consult the MS-DOS manual for more information on SHARE.EXE, or type 'HELP SHARE' at the command prompt (MS-DOS 5.0 or later only).

On an MS-DOS system with many programs loaded, you may find that adding SHARE consumes more lower memory area than is convenient. Consider running 'LOADHIGH SHARE' (on MS-DOS 5.0 or later only). Consult the MS-DOS manual for more information about LOADHIGH, or type 'HELP LOADHIGH' at the command prompt. On MS-DOS 6.2 or later, the MEMMAKER command can help you maximize real-mode memory usage. Run MEMMAKER after adding SHARE.EXE to 'AUTOEXEC.BAT' to help sort out what programs to load where. Consult the MS-DOS manual for more information on MEMMAKER, or type 'MEMMAKER /?' at the command prompt.

> *Warning:* If your system runs a third-party DOS extender (such as QEMM or 386MAX), please use the commands and memory load configuration tools provided by your memory manager, in place of LOADHIGH or MEMMAKER. Consult the manuals that come with your DOS extender for further information.

1.3 Typical initialization files

These are typical initialization files for use with the Cygnus Developer's Kit:

CONFIG.SYS:

```
REM 486/DX50 VL-bus
DEVICE=C:\DOS\SMARTDRV.EXE /double_buffer
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE RAM X=CC00-CFFF
REM EMM386 line excludes RAM area used by this specific network card.
BUFFERS=15,0
REM MEMmaker chose this setting for BUFFERS.
FILES=100
DOS=UMB
REM MEMMAKER under DOS 6.2 splits the HIGH, UMB commands.
LASTDRIVE=E
REM or whatever is your last drive.
FCBS=16,8
DEVICEHIGH /L:2,12048 =C:\DOS\SETVER.EXE
DOS=HIGH
REM MEMMAKER under DOS 6.2 splits the HIGH, UMB commands.
SHELL=C:\DOS\COMMAND.COM C:\DOS\ /E:1024 /p
REM The /E option leaves enough env space for Cygnus SETENV actions.
DEVICEHIGH /L:1,9072 =C:\DOS\ANSI.SYS
```

AUTOEXEC.BAT:

```
@ECHO OFF
LH /L:0;2,45488 /S C:\DOS\SMARTDRV.EXE
PATH C:\DOS;C:\WINDOWS;C:\BAT;C:\NET;C:\bin;C:\gnu;c:\utils
C:\DOS\MOUSE.COM
LH /L:2,13984 SHARE
PROMPT $p$g
SET TMP=C:\TEMP
REM Cygnus tools and other programs use TMP
SET TEMP=C:\TEMP
REM DOS and other programs use TEMP
SET LOGNAME=jax
REM RCS uses LOGNAME
LH /L:1,6384 C:\DOS\DOSKEY /INSERT
REM I love command line recall!
REM This might be a place to run C:\CYGNUS\SETENV.BAT
```

1.4 GO32, the 32-bit launcher

GO32 is the protected-mode 32-bit application launcher used by the tools in the Cygnus Developer's Kit. It works best on many MS-DOS machines if you use HIMEM.SYS and EMM386.EXE. If you use MEMMAKER, be sure to tell it you have programs which need expanded memory (GO32 uses expanded memory).

GO32 can also run Cygnus programs under versions of Windows, Windows for Workgroups, Windows NT, OS/2 and Desqview. Special versions of GO32 may be necessary for all but MS-DOS; if you need one of these special versions, please send a problem report with a problem category of

```
>Category: dos
```

and a synopsis of

```
>Synopsis: Need go32 support
```

1.5 Remote debugging using asynctsr

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.

The following sample session illustrates the steps needed to start a program under GDB control on a Hitachi SH chip, using a DOS host. The example uses a sample SH program called 't.x'. The procedure is similar for other chips. In general, you must run asynctsr and then mode before GDB.

First hook up your development board. In this example, we use a board attached to serial port COM2; if you use a different serial port, substitute its name in the argument of the mode command. When you call asynctsr, the auxiliary communications program used by the degugger, you give it just the numeric part of the serial port's name; for example, 'asyncstr 2' below runs asyncstr on COM2.

```
C:\SH\TEST> asynctsr 2

C:\SH\TEST> mode com2:9600,n,8,1,p

Resident portion of MODE loaded

COM2: 9600, n, 8, 1, p
```

> *Warning:* We have noticed a bug in PC-NFS that conflicts with asynctsr. If you also run PC-NFS on your DOS host, you may need to disable it, or even boot without it, to use asynctsr to control your development board.

For more information on remote debugging, see section "The GDB remote serial protocol" in *Debugging with GDB*.

1.6 Bug report form

The Cygnus bug-report template accompanies this release in the file '`C:\instdir\SEND_PR.TXT`'. Customize the Cygnus bug-report form by filling in your customer ID:

1. Find your customer ID in the cover letter that came with this release; or call the Cygnus hotline +1 415 903 1401 to ask what it is.
2. Run your favorite editor on '`C:\instdir\LIB\SEND_PR.TXT`'
3. Search for the text '`>Customer-Id: unknown`'
4. Replace the string '`unknown`' with your customer ID.

Copy this file into an email message and send it to `bugs@cygnus.com` or print it out and fax it to Cygnus Support.

2 Warnings for DOS Developer's Kits

See section "Limitations and Warnings" in *Release Notes*, for special considerations that apply to the Developer's Kit on any host. There are also a few special considerations for MS-DOS hosts only, which we describe here.

2.1 Memory requirements for MS-DOS

The toolkit uses up to 128Mb of extended memory if present, and up to 128Mb of disk space can be used for swapping. Disk caching programs speed compilation greatly.

We do not recommend using the cross-development kit with less than four (4) megabytes of RAM.

We provide a MS-DOS extender with the cross-development kit for MS-DOS which does swapping to disk when MS-DOS runs out of memory. To avoid excessive swapping you must have at least two (2) megabytes of RAM to run G++ on a PC with MS-DOS.

If you've got more than two megabytes, the extra memory can be used as a disk cache to significantly improve performance.

2.2 C++ compiler

There is no specific script which runs the C++ compiler. To use G++, simply run GCC with the '-lgxx' option. 'xx' is used because the character + is not a valid character for MS-DOS filenames.

2.3 Disk cache

While not strictly necessary to run the cross-compiler, using a good disk cache speeds up the compiler noticeably. SMARTdrive, for example, is a cache program that comes with MS-DOS from version 5 onwards; refer to your MS-DOS manual for details on how to use it.

2.4 Expanding environment space

If your environment is short on space for setting environment variables, you can expand it by editing the file 'CONFIG.SYS'. Edit the line

```
SHELL=C:\COMMAND.COM /P /E:envsiz
```

to reflect the new environment space size. *envsiz* stands for the environment size, in bytes.

2.5 Debugging AMD 29K programs

To develop software for the AMD 29k, you can use either the montip program to communicate with a board over a serial line, or the program isstip as a simulator. isstip and montip are both available from AMD at no charge. They are not included with this release.

To use the simulator, simply set the environment variable UDICONF to the full path of the file 'udiconfs.txt', as in the example

```
set UDICONF drive:instdir\AMD\BIN\UDICONFS.TXT
```

For instance, if you installed the Developer's Kit in 'C:\CYGNUS', indicate the full path as:

```
'C:\CYGNUS\AMD\BIN\UDICONFS.TXT'
```

Then place the simulator in resident memory with the command:

```
isstip isstip.exe
```

isstip takes information from the configuration file 'udiconfs.txt', which contains the following entries:

```
# name  command + options
iss     isstip.exe -r osboot
```

'osboot' is a file that is loaded into the isstip program. 'osboot', in this case, simulates ROMS, provides an interface for input and output, etc.

> *Warning:* once the isstip program is started, running the compiler hangs the machine. This is because isstip uses a different and incompatible DOS extender from the compiler, and thus the programs cannot coexist.

If you wish to use GDB with the target board, first change your working directory to the directory containing 'MONTIP.EXE' and type

```
montip montip.exe
```

before invoking gdb. See section "Getting In and Out of GDB" in *Debugging with* GDB, arguments to gdb.

Again, isstip and montip are both available from AMD at no charge. They are not included with this release.

2.6 DEL does not work in Info

GNU Info, the online documentation browser, is available with this release on our MS-DOS hosts.

Unfortunately, the DOS version of Info, INFO.EXE, does not recognize the DEL key. This key is normally used for paging backwards within a node in Info. As a workaround, you can page backwards in Info by typing ESC v.

2.7 Includes with conflicting filenames

> *Note:* the following information only matters when you want to read the Developer's Kit source code. You do not need to alter your programming style in any way. The C preprocessor handles this mapping automatically, based on the specifications in the file 'header.gcc'.

In the include directories of a standard GNU compiler system there are sometimes files whose names are too long to represent under MS-DOS and its related operating environments. There are also pairs of files where the only difference between the names is whether some characters are upper-case. For example, one of the standard C header files is called 'string.h', and one of the LIBG++ header files is called 'String.h'.

We handled this in your Cygnus toolkit by moving uppercase-distinct and too-long filenames to legal MS-DOS filenames in a subdirectory called

'upcase'. Here are three examples of how we rename files to avoid conflicts in the MS-DOS file system. These files are all from the 'include.cxx' subdirectory:

```
filename          conflict          file renamed
---------         --------          ------------
./iostreamP.h    ./iostream.h     ./upcase/iostream.h
./Complex.h      ./complex.h      ./upcase/Complex.h
./Regex.h        ./regex.h        ./upcase/Regex.h
```

Any include directory may contain an 'upcase' subdirectory. The header files in it would be in the parent directory on a Unix system. Again, you do not need to alter your programming style in any way; the compiler remaps the locations of these files, so that '#include <iostream.h>' works correctly even though it resides physically in a subdirectory of 'include.cxx'.

3 Hints and Common Problems

This section is part of a FAQ (frequently asked questions) list which was developed by Steve Chamberlain (sac@cygnus.com) to help with common DOS tool-chain problems. Please send comments or suggestions for further FAQ's to him.

Q: *How do I pass a command line of greater than 127 characters to any of the tools?*

A: DOS doesn't allow long command lines. But you can put your command line into a file (called a *response file*) and reference the file on the command line with a @ sign. Then the tool will read the file as if you typed it onto the command line. For example:

```
gcc @foo
```

reads the contents of the file 'foo' as if they were on the command line.

Q: *Can I use the tools on a 286?*

A: No.

Q: *Can I run it under Microsoft Windows?*

A: Yes. In fact, it works a bit faster under MS-Windows.

Q: *Why do some birds fly south for the winter?*

A: It's too far to walk.

Q: *Why does a tool say "Error: not enough memory to run go32!".*

A: This often happens when trying to run MAKE, but it can happen to any of the tools if you don't have enough memory. MAKE calls gcc, gcc calls cc1 or the assembler, and so on. Each invocation uses more memory. You can avoid this problem by not using MAKE, or stripping out unused

TSRs from your 'AUTOEXEC.BAT' or 'CONFIG.SYS' file so that there is more spare memory in the bottom 640K. Try running 'MEMMAKER'.

Use 'MEM' to see how much conventional memory you have free. You'll need around 500K to allow MAKE to run properly.

Q: *How do I get the online help?*

A: If your INFOPATH is set correctly then all you need do is type:

```
info
```

If you don't have INFOPATH set, you'll get an error like this:

```
info.exe: dir: file not found
```

You'll then need to specify the path on the command line like this:

```
info -d c:/cygnus/info
```

4 Technical info about the release

The binaries in this release were compiled on a Unix workstation with GCC generating 32-bit 386 code. The code is run with a DOS extender 'go32.exe' which uses the flat memory model available in the 386. Source for this extender is available on the net. Contact info@cygnus.com if you have problems obtaining it.

The files are stored on the floppies using Microsoft's 'COMPRESS' program. You can install files without using the install program by just copying them into the right place on your hard drive and running 'EXPAND'. Since the files are stored on the floppy using their full name (and not the marked as compressed by using Microsoft's '.XX_ ' naming convention) you'll have to use a temporary file.

```
copy b:\bin\cc1.exe c:\foo\bin
expand c:\foo\bin\cc1.exe c:\foo\bin\tmp
rename c:\foo\bin\tmp c:\foo\bin\cc1.exe
```

Programming Embedded Systems

With the Cygnus Developer's Kit

Cygnus Support

Copyright © 1993, 1994, 1995 Cygnus Support

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

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

Table of Contents

1 Hitachi SH Development 1
- 1.1 What to Call the Tools 1
- 1.2 Compiling for the Hitachi SH 1
- 1.3 Using C++ for the Hitachi SH 2
- 1.4 Register handling 3
- 1.5 Debugging code for the Hitachi SH 3
  - 1.5.1 Connecting to Hitachi boards 4
  - 1.5.2 Using the E7000 in-circuit emulator 5
- 1.6 Hitachi SH documentation 6

2 IDT MIPS Development 7
- 2.1 What to Call the Tools 7
- 2.2 Bootstrapping the Tools for IDT/MIPS Development 7
- 2.3 Compiler options for MIPS 10
  - 2.3.1 GCC options for code generation 10
  - 2.3.2 GCC options for floating point 11
  - 2.3.3 Floating point subroutines 12
  - 2.3.4 Linking with the GOFAST library 13
  - 2.3.5 Full compatibility with the GOFAST library . . . 14
  - 2.3.6 GCC options to avoid for IDT R3000 boards . . . 14
- 2.4 Predefined preprocessor macros 16
- 2.5 Assembling MIPS R3000 code 16
  - 2.5.1 Assembler options 16
  - 2.5.2 ECOFF object code 17
  - 2.5.3 Directives for debugging information 17
- 2.6 Remote IDT/MIPS Debugging 18
- 2.7 Configuring GNU source for IDT/MIPS 19
- 2.8 IDT/MIPS documentation 20
- 2.9 Some General Information 20
  - 2.9.1 Assembly with C preprocessing 20
  - 2.9.2 Useful listings from GNU `as` or GCC 20
  - 2.9.3 An extra initialization function 22
- 2.10 What to Call the Tools 22
- 2.11 Compiling for LynxOS 23
  - 2.11.1 Compiler options for LynxOS 23
  - 2.11.2 Default options for your environment 24
  - 2.11.3 Predefined preprocessor macros 25
- 2.12 LynxOS Debugging with GDB 25
  - 2.12.1 Multithread debugging on LynxOS 25
    - 2.12.1.1 Switching and inquiring on threads 26

2.12.1.2 Breakpoint features for LynxOS threads 27
2.12.1.3 Watchpoint limitations for LynxOS threads 28
2.12.2 Cross debugging with `gdbserver` 28
2.13 LynxOS Subroutine Libraries 30
2.14 Object formats supported 31
2.15 Configuring GNU source for LynxOS 31
2.16 LynxOS Documentation 32

3 NLM Development 33

3.1 What to Call the Tools 33
3.2 Compiling for NetWare 33
3.3 Compiler and Linker Options for NetWare 34
3.4 Making an NLM 34
3.4.1 Differences from DOS development tools 35
3.4.2 What goes in the *file*.def header 36
3.5 NetWare Debugging with GDB 40
3.6 Subroutine Libraries 41
3.7 Predefined Preprocessor Macros 41
3.8 Configuring GNU source for NetWare 42
3.9 NetWare Development Documentation 42

4 Fujitsu SPARClite Development 43

4.1 What to Call the Tools 43
4.2 Compiling for the SPARClite 43
4.2.1 Setting up GCC for the SPARClite 44
4.2.2 SPARC options for architecture and code generation 45
4.2.3 Compiler command-line options for floating point 46
4.2.4 Floating point subroutines 47
4.2.5 SPARC options for unfinished features 47
4.3 Assembling SPARClite code 47
4.4 Remote SPARClite Debugging with GDB 47
4.5 SPARClite documentation 48

1 Hitachi SH Development

1.1 What to Call the Tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The names are constructed by using as a prefix the '`--target`' argument to `configure`. For example, the compiler (called simply `gcc` in native configurations) is called by the name for Hitachi SH cross-development, `sh-hms-gcc`. Likewise, the SH-configured `gdb` is called by the name `sh-hms-gdb`.

For DOS-hosted toolchains, the tools are simply called by their standard names, e.g., `gcc`, `gdb`, etc.

1.2 Compiling for the Hitachi SH

Once the toolchain is configured to generate code for the SH, you can control variances in code generation directly from the command line.

Note: Much of the SH code is experimental, and may change in the future.

General Options

`-msh1` Generate code for the SH-1 chip. This is the default behavior for the SH configuration.

`-msh2` Generate code for the SH-2 chip.

There is also an option '`-msh3`' which will generate code for the SH-3 chip when the details are made available. This option currently yields the same results as '`-msh2`'.

Experimental Features

`-mfastcode`
: Generate fast code, rather than small code.

`-msmallcode`
: Generate small code, rather than fast code.

`-mnosave`
: Use a different calling convention. The registers 'R8', 'R9', 'R10', and 'R11' are all call-used with this setting (see Section 1.4 "Register handling," page 3).

`-mhitachi`
: Use Hitachi's calling convention rather than that for GCC. The registers 'MACH' and 'MACL' are saved with this setting (see Section 1.4 "Register handling," page 3).

`-mbsr`
: Use `bsr` calls to code backward-referenced within the function. *Note:* This may fail if the target is too far away.

`-mshortaddr`
: Assume that all static data fits into 16 bits, and keep pointers to them in word-length rather than long-length pointers.

`-mbigtable`
: Generate jump tables for switch statements using four-byte offsets rather than the standard two-byte offset. This option is necessary when the code within a switch statement is larger that 32k. If the option is needed and not supplied, the assembler will generate errors.

1.3 Using C++ for the Hitachi SH

This special release includes support for the C++ language. This support may in certain circumstances add up to 5K to the size of your executables

The new C++ support involves new startup code that runs C++ initializers before 'main()' is invoked. If you have a replacement for the file 'crt0.o' (or if you call 'main()' yourself) you must call '_main()' before calling 'main()'.

You may need to run these C++ initializers even if you do not write in C++ yourself. This could happen, for instance, if you are linking against a third-party library which itself was written in C++. You may not be able to tell that it was written in C++ because you are calling it with C entry points prototyped in a C header file. Without these initializers, functions written in C++ may malfunction.

If you are not using any third-party libraries, or are otherwise certain that you will not require any C++ constructors you may suppress them by adding the following definition to your program:

```
int __main() {}
```

1.4 Register handling

The first four words of arguments are passed in registers 'R4' through 'R7'. All remaining arguments are pushed onto the stack, last to first, so that the lowest numbered argument not passed in a register is at the lowest address in the stack. The registers are always filled, so a double word argument starting in 'R7' would have the most significant word in 'R7' and the least significant word on the stack.

When a function is compiled with the default options, it must return with registers 'R8' through 'R15' unchanged. Registers 'R0' through 'R7', 'T', 'MACH' and 'MACL' are volatile.

The '-mhitachi' switch makes the 'MACH' and 'MACL' registers caller-saved, which is compatible with the Hitachi tool chain at the expense of performance.

The '-mnosave' switch makes registers 'R0' through 'R11' volatile, which can often improve performance.

Note that functions compiled with different calling conventions cannot be run together without some care.

1.5 Debugging code for the Hitachi SH

GDB needs to know these things to talk to your Hitachi SH:

1. that you want to use 'target remote', the remote debugging interface for the Hitachi SH microprocessors, or 'target e7000', the in-circuit emulator for the Hitachi SH and the Hitachi 300H.
2. what serial device connects your host to your Hitachi board (the first serial device available on your host is the default).
3. if you are using a Unix host, what speed to use over the serial device.

1.5.1 Connecting to Hitachi boards

You can use the GDB remote serial protocol to communicate with a Hitachi SH board. You must first link your programs with the "stub" module 'src/gdb/config/sh/stub.c'. This module manages the communication with GDB. See section "The GDB remote serial protocol" in *Debugging with GDB*, for more details.

Use the special gdb command 'device *port*' if you need to explicitly set the serial device. The default *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. You must also use the DOS mode command to set up the serial port on the DOS side.

The following sample session illustrates the steps needed to start a program under GDB control on an SH, using a DOS host. The example uses a sample SH program called 't.x'. The procedure is the same for other Hitachi chips in the series.

First hook up your development board. In this example, we use a board attached to serial port COM2; if you use a different serial port, substitute its name in the argument of the mode command.

```
C:\SH\TEST> mode com2:9600,n,8,1,p

Resident portion of MODE loaded

COM2: 9600, n, 8, 1, p
```

Now that serial communications are set up, and the development board is connected, you can start up GDB. Call gdb with the name of your program as the argument. gdb prompts you, as usual, with the prompt '(gdb)'. Use two special commands to begin your debugging session: 'target hms' to specify cross-debugging to the Hitachi board, and the load command to download your program to the board. load displays the names of the program's sections, and a '*' for each 2K of data downloaded. (If you want to refresh GDB data on symbols or on the executable file without downloading, use the GDB commands file or symbol-file. These commands, and load itself, are described in section "Commands to specify files" in *Debugging with GDB*.)

```
C:\SH\TEST> gdb t.x
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
for details.
GDB 4.13-94q4, Copyright 1994 Free Software Foundation, Inc...
(gdb) target remote
Connected to remote SH HMS system.
(gdb) load t.x
.text   : 0x8000 .. 0xabde ***********
.data   : 0xabde .. 0xad30 *
.stack  : 0xf000 .. 0xf014 *
```

At this point, you're ready to run or debug your program. From here on, you can use all the usual GDB commands. The `break` command sets breakpoints; the `run` command starts your program; `print` or `x` display data; the `continue` command resumes execution after stopping at a breakpoint. You can use the `help` command at any time to find out more about GDB commands.

Remember, however, that *operating system* facilities aren't available on your development board; for example, if your program hangs, you can't send an interrupt—but you can press the RESET switch!

Use the RESET button on the development board:

- to interrupt your program (don't use `Ctrl-C` on the DOS host—it has no way to pass an interrupt signal to the development board); and
- to return to the GDB command prompt after your program finishes normally. The communications protocol provides no other way for GDB to detect program completion.

In either case, GDB sees the effect of a RESET on the development board as a "normal exit" of your program.

1.5.2 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 E7000:

`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.

The monitor command set makes it difficult to load large ammounts of data over the network without using FTP. We recommend you try not to issue `load` commands when communicating over Ethernet; use the `ftpload` command instead.

1.6 Hitachi SH documentation

The following manuals provide extensive documentation on the SH. They are produced by and available from Hitachi Microsystems; contact your friendly Field Application Engineer for details.

SH Microcomputer User's Manual
Semiconductor Design & Development Center, 1992

Hitachi SH2 Programming Manual
Semiconductor and Integrated Circuit Division, 1994

2 IDT MIPS Development

2.1 What to Call the Tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The names are constructed by using, as a prefix, the name of the configured target (i.e., the string specified with '`--target`' to `configure`). For example, the compiler (called simply `gcc` in native configurations) is called by one of these names:

`mips-idt-ecoff-gcc`
: *If configured for big-endian byte ordering.*

`mipsel-idt-ecoff-gcc`
: *If configured for little endian byte ordering.*

2.2 Bootstrapping the Tools for IDT/MIPS Development

Before you can use the Cygnus Developer's Kit to build your programs for IDT boards, you need a C library and C run-time initialization code. Unless you already have suitable libraries of your own, you must integrate the Cygnus C libraries with low-level code supplied by IDT. This low-level code initializes the C run-time environment, and describes the hardware interface to the Cygnus C libraries.

To begin with, make sure you have the following C and assembly source files from IDT:

C source files:

```
drv_8254.c          sys.c
idt_int_hand.c      syscalls.c
idtfpip.c           timer_int_hand.c
sbrk.c
```

C header files:

```
dpac.h              idtio.h
excepthdr.h         idtmon.h
fpip.h              iregdef.h
i8254.h             saunder.h
idt_entrypt.h       setjmp.h
idtcpu.h
```

Assembler files:

idt_csu.S	lnkexit.S
idt_except.S	lnkhelp.S
idtfpreg.S	lnkinstal.S
idtmem.S	lnkio.S
idttlb.S	lnkioctl.S
idtwbf.S	lnkjmp.S
lnkatb.S	lnkmem.S
lnkcach.S	lnknimp.S
lnkchar.S	lnkprint.S
lnkcio.S	lnksbrk.S
lnkcli.S	lnkstr.S

Then follow these steps to integrate the low-level IDT code with your Cygnus Developer's Kit:

1. IDT supplies the C run-time initialization code in the file 'idt_csu.S'. Since GNU CC expects to find the initialization module under the name crt0.o, rename the source file to match:

```
$ mv idt_csu.S crt0.S
```

2. Edit the contents of 'crt0.S'. A few more instructions are needed to ensure correct initialization, and to ensure that your programs exit cleanly. At the end of the file (after a comment including the text 'END I/O initialization'), look for these lines:

```
        jal      main

ENDFRAME(start)
```

   Insert 'move ra,zero' before 'jal main' to mark the top of the stack for the debugger, and add two lines after the call to main to call the exit routine (before the 'ENDFRAME(start)'), so that the end of the file looks like this:

```
        move    ra,zero
        jal     main

        move    a0,v0
        jal     exit

ENDFRAME(start)
```

3. Edit 'syscalls.c', the interface to the low-level routines required by the C library, to remove the leading underbar from two identifiers:

   a. Rename `_kill` to `kill`;
   b. Rename `_getpid` to `getpid`.

4. Edit '`lnksbrk.S`' to remove the definition of `_init_sbrk`; this definition is not needed, since it is available in '`sbrk.c`'. Delete the lines marked with '`-`' at the left margin below:

```
        .text

-FRAME(_init_sbrk,sp,0,ra)
-       j       ra
-ENDFRAME(_init_sbrk)
-
-
-
 FRAME(_init_file,sp,0,ra)
        j       ra
 ENDFRAME(_init_file)
```

5. Use your Cygnus Developer's Kit to assemble the '`.S`' files, like this (use the compiler driver `gcc` to permit C preprocessing).

   For concreteness, these example commands assume the `mips` *(big-endian) variant of the configuration; if you ordered tools configured for little-endian object code, type '*`mipsel`*' wherever the examples show '*`mips`*'.*

```
$ mips-idt-ecoff-gcc -g -c *.S
```

6. Compile the '`.c`' files.

   One particular C source file, '`drv_8254.c`' requires two special preprocessor symbol definitions: '`-DCLANGUAGE -DTADD=0xBF800000`'. *Be careful to type the constant value for '*`TADD`*' accurately;* the correct value is essential to allow the IDT board to communicate over its serial port.

   The two special preprocessor definitions make no difference to the other C source files, so you can compile them all with one call to the compiler, like this:

```
$ mips-idt-ecoff-gcc -g -O \
   -DCLANGUAGE -DTADD=0xBF800000 -c *.c
```

   (The example is split across two lines simply due to formatting constraints; you can type it on a single line instead of two lines linked by a '\', of course.)

7. Add the new object files to the C library archive, 'libc.a', from your Cygnus Developer's Kit. Assuming you installed the Kit in '/usr/cygnus/' as we recommend:

```
$ mips-idt-ecoff-ar rvs /usr/cygnus/progressive-94q1/\
H-host/mips-idt-ecoff/lib/libc.a *.o
```

   As before, you can omit the '\' and type a single line. 'H-*host*' stands for the string that identifies your host configuration; for example, on a SPARC computer running SunOS 4.1.3, you'd actually type 'H-sparc-sun-sunos4.1.3'.

2.3 Compiler options for MIPS

When you run GCC, you can use command-line options to choose machine-specific details. For information on all the GCC command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

2.3.1 GCC options for code generation

-mcpu=r3000
-mcpu=*cputype*

Since the IDT boards are based on the MIPS R3000, the default for this particular configuration is '-mcpu=r3000'.

In the general case, use this option on any MIPS platform to assume the defaults for the machine type *cputype* when scheduling instructions. The default *cputype* on other MIPS configurations is 'default', which picks the longest cycle times for any of the machines, in order that the code run at reasonable rates on any MIPS CPU. Other choices for *cputype* are 'r2000', 'r3000', 'r4000', and 'r6000'. While picking a specific *cputype* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (instruction set architecture) unless you use the '-mips2' or '-mips3' switch.

-mgpopt
-mno-gpopt

With '-mgpopt', write all of the data declarations before the instructions in the text section. This allows the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default when you compile with optimization.

`-mstats`
`-mno-stats`
: With '`-mstats`', for each non-inline function processed, emit one line to the standard error file to print statistics about the program (number of registers saved, stack size, etc.).

`-mmemcpy`
`-mno-memcpy`
: With '`-mmemcpy`', make all block moves call `memcpy` (a C library string function) instead of possibly generating inline code.

`-mlong-calls`
`-mno-long-calls`
: Do all calls with the `JALR` instruction, which requires loading up a function's address into a register before the call. You need this switch if you call functions outside of the current 512 megabyte segment (unless you use function pointers for the call).

`-mhalf-pic`
`-mno-half-pic`
: Put pointers to extern references into the data section and load them up, rather than putting the references in the text section.

`-G` *num*
: Put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (*gp* or *$28*), instead of the normal two words used. By default, *num* is 8. When you specify another value, `gcc` also passes the '`-G` *num*' switch to the assembler and linker.

2.3.2 GCC options for floating point

These options select software or hardware floating point.

`-msoft-float`
: Generate output containing library calls for floating point. The '`mips-idt-ecoff`' configuration of '`libgcc`' (an auxiliary library distributed with the compiler) include a collection of subroutines to implement these library calls.

In particular, this GCC configuration generates subroutine calls compatible with the US Software "GOFAST R3000" floating point library, giving you the opportunity to use either the

'libgcc' implementation or the US Software version. IDT includes the GOFAST library in their IDT C 5.0 package; you can also order libraries separately from IDT as the "IDT KIT". See Section 2.3.4 "Linking with the GOFAST library," page 13, for examples of how to use GCC to link with the GOFAST library.

To use the 'libgcc' version, you need nothing special; GCC links with 'libgcc' automatically after all other object files and libraries.

Because the calling convention on MIPS architectures depends on whether or not hardware floating-point is installed, '-msoft-float' has one further effect: GCC looks for subroutine libraries in a subdirectory 'soft-float', for any library directory in your search path. (*Note*: This does not apply to directories specified using the '-l' option.) With the Cygnus Developer's Kit, you can select the standard libraries as usual with '-lc' or '-lm', because the 'soft-float' versions are installed in the default library search paths.

> *Warning:* Treat '-msoft-float' as an "all or nothing" proposition. If you compile any module of a program with '-msoft-float', it's safest to compile all modules of the program that way—and it's essential to use this option when you link.

`-mhard-float`

Generate output containing floating point instructions, and use the corresponding MIPS calling convention. This is the default.

2.3.3 Floating point subroutines

Two kinds of floating point subroutines are useful with GCC:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.

   When you indicate that no hardware floating point is available (with the GCC option '-msoft-float', GCC generates calls compatible with the GOFAST library, proprietary licensed software available from US Software. If you do not have this library, you can still use software floating point; 'libgcc', the auxiliary library distributed with GCC, includes compatible—though slower—subroutines.

2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

2.3.4 Linking with the GOFAST library

The GOFAST library is available with two interfaces; GCC '-msoft-float' output places all arguments in registers, which (for subroutines using double arguments) is compatible with the interface identified as "Interface 1: all arguments in registers" in the GOFAST documentation. For full compatibility with all GOFAST subroutines, you need to make a slight modification to some of the subroutines in the GOFAST library. See Section 2.3.5 "Full compatibility with the GOFAST library," page 14, for details.

If you purchase and install the GOFAST library, you can link your code to that library in a number of different ways, depending on where and how you install the library.

To focus on the issue of linking, the following examples assume you've already built object modules with appropriate options (including '-msoft-float').

This is the simplest case; it assumes that you've installed the GOFAST library as the file 'fp.a' in the same directory where you do development, as shown in the GOFAST documentation:

```
$ mips-idt-ecoff-gcc -o prog prog.o ... -lc fp.a
```

In a shared development environment, this example may be more realistic; it assumes you've installed the GOFAST library as '*ussdir*/libgofast.a', where *ussdir* is any convenient directory on your development system.

```
$ mips-idt-ecoff-gcc -o program program.o ... -lc -Lussdir -lgofast
```

Finally, you can eliminate the need for a '-L' option with a little more setup, using an environment variable like this (the example assumes you use a command shell compatible with the Bourne shell):

```
$ LIBRARY_PATH=ussdir; export LIBRARY_PATH

$ mips-idt-ecoff-gcc -o program program.o ... -lc -lgofast
```

As for the previous example, the GOFAST library here is installed in '*ussdir*/libgofast.a'. The environment variable LIBRARY_PATH instructs GCC to look for the library in *ussdir*. (The syntax shown here for setting the environment variable is the Unix Bourne Shell syntax; adjust as needed for your system.)

Notice that all the variations on linking with the GOFAST library explicitly include '`-lc`' *before* the GOFAST library. '`-lc`' is the standard C subroutine library; normally, you don't have to specify this, since linking with that library is automatic.

When you link with an alternate software floating-point library, however, the order of linking is important. In this situation, specify '`-lc`' *to the left* of the GOFAST library, to ensure that standard library subroutines also use the GOFAST floating-point code.

2.3.5 Full compatibility with the GOFAST library

The GCC calling convention for functions whose first and second arguments have type `float` is not completely compatible with the definitions of those functions in the GOFAST library, as shipped.

These functions are affected:

```
fpcmp   fpadd   fpsub   fpmul   fpdiv   fpfmod
fpacos  fpasin  fpatan  fpatan2 fppow
```

Since the GOFAST library is normally shipped with source, you can make these functions compatible with the GCC convention by adding this instruction to the beginning of each affected function, then rebuilding the library:

```
        move    $5,$6
```

2.3.6 GCC options to avoid for IDT R3000 boards

These options are meant for other forms of the MIPS architecture:

`-mabicalls`
`-mno-abicalls`
: Emit (or do not emit) the assembler directives '`.abicalls`', '`.cpload`', and '`.cprestore`' that some System V.4 ports use for position independent code.

`-mips2`
: Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions).

`-mmips-as`
: Generate code for the MIPS assembler. This is the default for many other MIPS platforms, but it requires an auxiliary program `mips-tfile` to encapsulate debugging information.

: `mips-tfile` *is not included* in your Cygnus Developer's Kit, since it is not required for the GNU assembler.

`-mrnames`
`-mno-rnames`
: Generate code using the MIPS software names for the registers, instead of the hardware names (for example, *a0* instead of *$4*). The converse '`-mno-rnames`' switch is the default.

> *Warning:* The GNU assembler will not build code generated with the '`-mrnames`' switch.

These options are harmless—but unnecessary—on the R3000:

`-mfp32`
: Assume that there are 32 32-bit floating point registers. This is the default.

`-mgas`
: Generate code for the GNU assembler. This is the default when GCC is correctly configured for this platform, using the '`-with-gnu-as`' configuration parameter.

`-mno-mips-tfile`
: The '`-mno-mips-tfile`' option prevents postprocessing the object file with the `mips-tfile` program, after the MIPS assembler has generated it to add debug support. The GNU assembler does not require `mips-tfile` in any case.

  `mips-tfile` *is not included* in your Cygnus Developer's Kit.

`-nocpp`
: Tell the MIPS assembler to avoid running the C preprocessor over user assembler files (with a '`.s`' suffix) when assembling them. The GNU assembler never runs the C preprocessor in any case.

  See Section 2.9.1 "Assembly with C preprocessing," page 20, for information about how to use assembly code that requires C-style preprocessing.

Avoid these options—although they appear in the configuration files, their implementation is not yet complete.

```
-mfp64
-mint64
-mips3
-mlong64
-mlonglong128
```

2.4 Predefined preprocessor macros

GCC defines the following preprocessor macros for the IDT/MIPS configurations:

Any MIPS *architecture:*
    `mips`, `_mips`, `__mips__`, `___mips__`, `__mips`, `___mips`.

MIPS *architecture with big-endian numeric representation:*
    `MIPSEB`, `_MIPSEB`, `__MIPSEB__`, `___MIPSEB__`, `__MIPSEB`, `___MIPSEB`.

MIPS *architecture with little-endian numeric representation:*
    `MIPSEL`, `_MIPSEL`, `__MIPSEL__`, `___MIPSEL__`, `__MIPSEL`, `___MIPSEL`.

r3000 MIPS *processor:*
    `R3000`, `_R3000`, `__R3000__`, `___R3000__`, `__R3000`, `___R3000`.

2.5 Assembling MIPS R3000 code

You should use GNU `as` to assemble GCC output. To ensure this, GCC should be configured using the '`--with-gnu-as`' switch (as it is in Cygnus distributions; see Section 2.7 "Configuring GNU source for IDT/MIPS," page 19). Alternatively, you can invoke GCC with the `-mgas` option.

GNU `as` for MIPS architectures supports the MIPS R2000 and R3000 processors.

2.5.1 Assembler options

The MIPS configurations of GNU `as` support three special options, and accept one other for command-line compatibility. See section "Command-Line Options" in *Using* `as`, for information on the command-line options available with all configurations of the GNU assembler.

`-G` *num*
    This option sets the largest size of an object that will be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format. The default value is 8.

`-EB`
`-EL`
    Any MIPS configuration of `as` can select big-endian or little-endian output at run time (unlike the other GNU development tools, which must be configured for one or the other). Use '`-EB`' to select big-endian output, and '`-EL`' for little-endian.

`-nocpp` This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With GNU `as`, there is no need for '`-nocpp`', because the GNU assembler itself never runs the C preprocessor. (If you have assembly code that requires C-style preprocessing, see Section 2.9.1 "Assembly with C preprocessing," page 20.)

2.5.2 ECOFF object code

Assembling for a MIPS ECOFF target supports some additional sections besides the usual `.text`, `.data` and `.bss`. The additional sections are `.rdata`, used for readonly data, `.sdata`, used for small data, and `.sbss`, used for small common objects.

When assembling for ECOFF, the assembler uses the `$gp` (`$28`) register to form the address of a "small object". Any object in the `.sdata` or `.sbss` sections is considered "small" in this sense. For external objects, or for objects in the `.bss` section, you can use the GCC '`-G`' option to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller will use `$gp`. Passing '`-G 0`' to `as` prevents it from using the `$gp` register on the basis of object size (but the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case). The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it. The size of an external object may be set with the `.extern` directive. For example, '`.extern sym,4`' declares that the object at `sym` is 4 bytes in length, whie leaving `sym` otherwise undefined.

Using small ECOFF objects requires linker support, and assumes that the `$gp` register has been correctly initialized (normally done automatically by the startup code). MIPS ECOFF assembly code must not modify the `$gp` register.

2.5.3 Directives for debugging information

MIPS ECOFF `as` supports several directives used for generating debugging information which are not support by traditional MIPS assemblers. These are `.def`, `.endef`, `.dim`, `.file`, `.scl`, `.size`, `.tag`, `.type`, `.val`, `.stabd`, `.stabn`, and `.stabs`. The debugging information generated by the three `.stab` directives can only be read by GDB, not by traditional MIPS debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers! See section "Assembler Directives" in *Using* `as`, for full information on all GNU `as` directives.

2.6 Remote IDT/MIPS Debugging

`mips-idt-ecoff-gdb` uses the MIPS remote serial protocol to connect your development host machine to the target board. On the target board itself, the IDT program `IDT/sim` implements the same protocol. (`IDT/sim` runs automatically whenever the board is powered up.)

Use these 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, this sequence connects to the target board through a serial port, and loads and runs a program called *prog* through the debugger:

```
host$ mips-idt-ecoff-gdb prog
GDB is free software and . . .
(gdb) target mips /dev/ttyb
(gdb) load
(gdb) run
```

`target mips` *hostname*`:`*portnumber*

If your GDB is configured to run from a SunOS or SGI host, you can specify a TCP connection instead of a serial port, using the syntax *hostname*`:`*portnumber* (assuming your IDT board is connected so that this makes sense; for instance, to a serial line managed by a terminal concentrator).

GDB also supports these special commands for IDT/MIPS targets:

`set mipsfpu off`

If your target board does not support the MIPS floating point coprocessor, you should use the command '`set mipsfpu off`' (you may wish to put this 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.

`set remotedebug n`
: 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 will be displayed. If you set it to 2 every character will be displayed. You can check the current value at any time with the command '`show remotedebug`'.

2.7 Configuring GNU source for IDT/MIPS

Your Cygnus Developer's Kit includes precompiled, ready-to-run binaries, with all defaults configured for IDT MIPS boards.

However, you may have occasion to reconfigure or rebuild the source; after all, improving your tools is one of your privileges as a free software user!

The script `configure` is used to specify many prearranged kinds of variations in the source. See section "What `configure` does" in *Cygnus configure*, for an overview of the source configuration process.

In particular, to configure for the MIPS environment, you should use these `configure` options:

- Specify '`--target=mips-idt-ecoff`' to generate or manage code for IDT/MIPS boards, with big-endian numeric representation.
- Specify '`--target=mipsel-idt-ecoff`' to generate or manage code for IDT/MIPS boards, with little-endian numeric representation.
- Be sure to specify '`--with-gnu-as`'. This avoids an incompatibility between the GNU compiler and the MIPS assembler. The MIPS assembler does not understand debugging directives. If you configure GCC without this option, the compiler requires a special program, `mips-tfile`, to generate these debugging directives. GNU `as` parses the debugging directives directly, and does not require `mips-tfile`.
- You may also wish to use '`--with-gnu-ld`', which will improve efficiency, or '`--with-stabs`', which will generate better debugging information. Note that only `gdb` can read this form of debugging information.

2.8 IDT/MIPS documentation

For information about the MIPS instruction set, see *MIPS RISC Architecture*, by Kane and Heindrich (Prentice-Hall).

For information about IDT's `IDT/sim` board monitor program, see *IDT/sim 4.0 Reference Manual* (IDT#703-00146-001/A).

For information about US Software's floating point library, see *U S Software GOFAST R3000 Floating Point Library* (United States Software Corporation).

2.9 Some General Information

The following sections give pointers to additional information of interest to IDT/MIPS board users.

2.9.1 Assembly with C preprocessing

There are two convenient options to assemble hand-written files that require C-style preprocessing. Both options depend on using the compiler driver program, `gcc`, instead of calling the assembler directly.

1. Name the source file using the extension '`.S`' (capitalized) rather than '`.s`'. `gcc` recognizes files with this extension as assembly language requiring C-style preprocessing.
2. Specify the "source language" explicitly for this situation, using the `gcc` option '`-xassembler-with-cpp`'.

2.9.2 Useful listings from GNU `as` or GCC

If you invoke `as` via the GNU C compiler (version 2), you can use the '`-Wa`' option to pass arguments through to the assembler. One common use of this option is to exploit the assembler's listing features. Assembler arguments you specify with `gcc -Wa` must be separated from each other (and the '`-Wa`') by commas. For example, the '`-alh`' assembler option in the following commandline:

```
$ mips-idt-ecoff-gcc -c -g -O -Wa,-alh,-L file.c
```

requests a listing with high-level language and assembly language interspersed.

The example also illustrates two other convenient options to specify for assembler listings:

1. The compiler debugging option '`-g`' is essential to see interspersed high-level source statements, since without debugging information the assembler cannot tie most of the generated code to lines of the original source file.
2. The additional assembler option '`-L`' preserves local labels, which may make the listing output more intelligible to humans.

These are the options to enable listing output from the assembler. By itself, '`-a`' requests listings of high-level language source, assembly language, and symbols.

You can use other letters to select specific options for the list: '`-ah`' requests a high-level language listing, '`-al`' requests an output-program assembly listing, and '`-as`' requests a symbol table listing. High-level listings require that a compiler debugging option like '`-g`' be used, and that assembly listings ('`-al`') be requested also.

You can use the '`-ad`' option to *omit* debugging directives from the listing.

When you specify one of these options, you can further control listing output and its appearance using these *listing-control* assembler directives:

`.nolist` Turn off listings from this point on.

`.list` Turn listings back on from here.

`.psize` *`linecount`* `,` *`columnwidth`*
: Describe the page size for your output. (Default 60, 200.) The assembler generates form feeds after printing each group of *`linecount`* lines. To avoid these automatic form feeds, specify 0 as the *`linecount`*.

`.eject` Skip to a new page (issue a form feed).

`.title` Use *`heading`* as the title (second line, immediately after the source file name and pagenumber).

`.sbttl` Use *`subheading`* as the subtitle (third line, immediately after the title line) when generating assembly listings.

If you do not request listing output with one of the '`-a`' options, these listing-control directives have no effect. You can also use the '`-an`' option to turn off all forms processing.

The letters after '`-a`' may be combined into one option, *e.g.*, '`-aln`'.

2.9.3 An extra initialization function

When you compile C or C++ programs with GNU C, the compiler quietly inserts a call at the beginning of `main` to a GCC support subroutine called `__main`. Normally this is invisible—but you may run into it if you want to avoid linking to the standard libraries, by specifying the compiler option '`-nostdlib`'. Include '`-lgcc`' at the end of your compiler command line to resolve this reference. This links with the compiler support library '`libgcc.a`', but putting it at the end of your command line ensures that you have a chance to link first with any special libraries of your own.

`__main` is the initialization routine for C++ constructors. Because GNU C is designed to interoperate with GNU C++, even C programs must have this call: otherwise C++ object files linked with a C `main` might fail.

2.10 What to Call the Tools

Aside from considerations of CPU architecture, there are two forms of the Cygnus Developer's Kit tools for LynxOS systems. You can run the development tools *native*, that is directly on a Lynx system; or you can run them as *cross-development* tools, using another system as the development *host* while generating or managing code for the Lynx *target*.

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory. Native tools, on the other hand, are called by simple names like `gcc` (the compiler) or `gdb` (the debugger).

Native development

On native configurations, you still need to be careful that your execution path is set up to get the right *versions* of the development tools. For example, LynxOS itself is distributed with '`/bin/gcc`' installed—but that's a very old release (version 1) of GNU CC. The tools from the Cygnus Developer's Kit are normally installed so that you can find them in '`/usr/progressive/bin`'. (See the Cygnus *Installation Notes* for LynxOS 2.2, for details on how to set up the Cygnus Developer's Kit installation, and suggestions to keep execution paths simple.) For example, you should get output like this from the system utility `which`:

```
$ which gcc
/usr/progressive/bin/gcc
```

You can also run GNU CC with the '-v' option to make sure you're running a version 2 compiler.

Cross development

The tool names for cross-development are constructed by using, as a prefix, the name of the configured target (i.e., the string specified with '--target' to the configure script). For example, the cross-compiler for LynxOS is called:

i386-lynx-gcc
: *If configured to generate code for Intel 386 architectures.*

m68k-lynx-gcc
: *If configured to generate code for Motorola 680x0 architectures.*

sparc-lynx-gcc
: *If configured to generate code for* SPARC *architectures.*

2.11 Compiling for LynxOS

The GNU compiler has a variety of options to cover many needs. This note discusses the options specifically intended for use with LynxOS. For information on all the GCC command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

2.11.1 Compiler options for LynxOS

When you run GCC, you can use these command-line options to choose some details specific to LynxOS. There are also compiler options specific to the machine architecture; see section "M680x0 Options" in *Using GNU CC*, and section "Intel 386 Options" in *Using GNU CC*.

-mposix
: Use the Posix-compatible version of the Lynx C library, and define the preprocessor macro _POSIX_SOURCE. (This is similar to the effect of '-X' with version 1 of GCC as distributed with LynxOS.)

-msystem-v
: Use this option for backward compatibility: it selects header files and libraries compatible with Unix System V release 3, and marks the output files with a "magic number" that identifies them as System V compatible. (This is similar to the effect of '-V' with version 1 of GCC as distributed with LynxOS.)

Warning: if you use this option, you may not use '-mthreads', '-p', or '-pg'. If you specify '-mposix' together with '-msystem-v', '-mposix' is ignored.

-mthreads
: Use alternate versions of system libraries that support multi-thread programming, and define the preprocessor macro _MULTITHREADED. (This is similar to the effect of '-m' with version 1 of GCC as distributed with LynxOS.)

In the older version of GCC distributed with LynxOS, there was also a '-k' option to specify COFF object format. No option is needed for this purpose with the Cygnus Developer's Kit version of the compiler, since the output format is always COFF.

2.11.2 Default options for your environment

If you always invoke GCC with a particular combination of options, you can collect these options in the environment variable GCC_DEFAULT_OPTIONS instead of listing them on the command line each time.[1]

Warning: The compiler developers at Cygnus recommend you *avoid* this environment variable, since it adds one more layer of complexity to using the compiler. In particular, its use may lead to these problems:

1. The relative order of these options must be fixed, and will always be wrong for some purposes. With the current implementation, options from GCC_DEFAULT_OPTIONS are always first on the command line (so that you can override them). This means that you cannot usefully include library options (such as '-lm'), since you need to put them at the end of the command line.
2. If you set the environment variable and forget about it, the compiler's behavior may be mysterious.
3. Bugreports may take longer to resolve, since the environment variable setting is one more important datum that may be accidentally omitted from bug reports.

---

[1] The Cygnus tools do *not* recognize these environment variables used by older ports of the GNU tools to LynxOS: COFFLD, SYSVCC, POSIXCC.

2.11.3 Predefined preprocessor macros

GCC defines the following preprocessor macros for LynxOS configurations:

Any LynxOS system:
: `unix`, `__unix__`, `__unix`, `Lynx`, `__Lynx__`, `__Lynx`, `IBITS32`, `__IBITS32__`, `__IBITS32`

Motorola 680x0 systems:
: `mc68000`, `M68K`, `__mc68000__`, `__M68K__`, `__mc68000`, `__M68K`

Intel 80x86 systems:
: `i386`, `I386`, `__i386__`, `__I386__`, `__i386`, `__I386`,

SPARC systems:
: `sparc`, `__sparc__`, `__sparc`

2.12 LynxOS Debugging with GDB

Two aspects of using the GNU debugger can differ significantly on LynxOS from many other systems. First, you can debug multithread programs; second, you may want to use the `gdbserver` program for cross-debugging.

2.12.1 Multithread debugging on LynxOS

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- '`thread` *threadno*', a command to switch among threads
- '`info threads`', a command to inquire about existing threads
- thread-specific breakpoints

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

Whenever GDB detects a new thread in your program, it displays the LynxOS identification for it with a message like this:

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

2.12.1.1 Switching and inquiring on threads

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

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

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

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

  For example,

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

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

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

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 process` *`n`* `thread` *`m`*`]`' to identify the thread.

2.12.1.2 Breakpoint features for LynxOS threads

You can choose whether to set breakpoints on all threads, or on a particular thread of your program.

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

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 GDB's numeric thread identifiers, shown in the first column of the '`info threads`' display.

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

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

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

Whenever your program stops under GDB for any reason, *all* threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, and to switch 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 LynxOS, rather than 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.

2.12.1.3 Watchpoint limitations for LynxOS threads

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 the same thread will remain current), then you can use watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression.

2.12.2 Cross debugging with `gdbserver`

When a LynxOS system is configured for production real-time use, the tradeoffs required may make it more convenient to do as much development work as possible on another system, for example by cross-compiling. You can make a similar choice with the GNU debugger, using an auxiliary program called `gdbserver`.

`gdbserver` is a control program for Unix-like systems, which allows you to connect your LynxOS program with a GDB that runs on another machine. On the remote machine, you use the GDB command `target remote` to connect to the LynxOS system.[2]

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

On the LynxOS (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`':

[2] `target remote` is otherwise used to debug bare-board systems, in which case it requires linking in special *stub* subroutines with your programs; with `gdbserver`, no such special stubs are needed.

```
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:

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

The only difference from the first 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. If you choose a port number that conflicts with another service, gdbserver prints an error message and exits). You must use the same port number with the host GDB target remote command.

On the host,

you need an unstripped copy of your program, since GDB needs symbols and debugging information. Start up GDB as usual, using the name of the local copy of your program as the first argument. (You may also need the '--baud' option if the serial line is running at anything other than 9600 bps.) After that, use target remote to establish communications with gdbserver. Its argument is either a device name (usually a serial device, like '/dev/ttyb'), or a TCP port descriptor in the form *host*:*port*. For example:

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

communicates with the server via serial line '/dev/ttyb', and

```
(gdb) 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'.

2.13 LynxOS Subroutine Libraries

One of the major effects of the compiler command-line options '-mposix', '-msystem-v', and '-mthreads' is to choose versions of the C subroutine libraries.

The compiler links by default with the standard Cygnus C subroutine library. See *The Cygnus C Support Library* for details on that library. Since LynxOS is a complete system, however, you have no need to define "stub" routines (see section "System Calls" in *The Cygnus C Support Library*); LynxOS provides the necessary system calls.

Your Cygnus Developer's Kit also includes a mathematical subroutine library. See *The Cygnus C Math Library* for more information.

If you specify '-msystem-v', the compiler does *not* use the Cygnus libraries; instead, your code links with the LynxOS System V.3 compatibility library.

If you specify '-mposix', you get *both* Posix-compatible subroutine libraries from LynxOS, and the standard Cygnus libraries. The Posix compatibility library overrides the Cygnus library for any subroutines present in both.

> *Warning:* None of these libraries are called 'libc.a', but LynxOS does ship with an older library (still present for compatibility) with that name. This means that *if you specify '-lc' explicitly, you should specify it last* on the compiler's command line. You may not need '-lc' at all, but some LynxOS releases include additional subroutines for special purposes in this library.

In most situations, you should use one of the '-m' options rather than specifying the libraries directly with '-l'. However, if you must insert your own library to override some of the system or Cygnus libraries, you may have to specify the equivalent series of '-l' commands explicitly. Here are the equivalences:

default For consistency with the libraries used by default (with no '-m' options), specify these libraries. The 'stdc' and 'stdm' libraries are the Cygnus C library and the Cygnus mathematical subroutine library, respectively.

```
-llynx -lstdc -lstdm -lstub
```

-mposix If you want the Posix compliant library, use a library list like this to allow the 'posix1' library to override parts of the Cygnus libraries:

```
-llynx -lposix1 -lstdc -lstdm -lstub
```

`-msystem-v`
: For the System V release 3 compatible library, use '`-lc_v`'.

`-mthreads`
: This option does not imply additional libraries; instead, it uses an alternate library search path to find different versions of the same libraries.

2.14 Object formats supported

The GNU compiler and assembler produce COFF format object code for the standard Lynx configurations. However, since older LynxOS tools generated a.out object code, the linker, debugger, and binary utilities are still able to read a.out object code format. This allows you to use and manage older libraries and object modules, and even to debug older complete programs, regardless of the change in object code format.

2.15 Configuring GNU source for LynxOS

Your Cygnus Developer's Kit includes precompiled, ready-to-run binaries, with all defaults configured for your LynxOS system.

However, you may have occasion to reconfigure or rebuild the source. For example, you may want to rebuild after receiving a bugfix in source-patch form.

You can use the script `configure` to specify many prearranged kinds of variations in the source. See section "Rebuilding From Source" in *Release Notes*, for a discussion of the source configuration process.

To configure for the LynxOS environment, you should use one of these `configure` options:

`--target=i386-lynx`
: To build code for LynxOS on Intel 386-based systems.

`--target=m68k-lynx`
: To build code for LynxOS on MVME147 or MVME167 boards.

`--target=sparc-lynx`
: To build code for LynxOS on SPARC-based systems.

Warning: The Free Software Foundation's Internet distributions of GCC support another Lynx configuration, because the GCC distribution is available independently of the other GNU tools. If you configure and rebuild GCC alone, also specify these options to `configure`:

`--with-gnu-as`
: Allow GCC to generate code suitable for the current GNU assembler, rather than restricting its output to the assembler found on LynxOS by default.

`--with-gnu-ld`
: Allow GCC to use the current GNU linker, rather than the linker found on LynxOS by default.

If you reconfigure and rebuild the entire tool chain as distributed by Cygnus, rather than GCC alone, these two options are applied to the compiler configuration automatically.

2.16 LynxOS Documentation

For general information about programming in LynxOS, see the *LynxOS Application Writer's Guide*.

For compatibility information about the alternative subroutine libraries on your LynxOS system, see the "Unix Compatibility" chapter in *LynxOS User's Manual: Volume 2, Supplementary Guides & Documents*.

Both documents are available from Lynx Real-Time Systems Inc., 16780 Lark Ave., Los Gatos, California 95030.

3 NLM Development

3.1 What to Call the Tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target configuration, so that you can install more than one set of tools in the same binary directory. These names are constructed by using, as a prefix, the name of the configured target. For example:

`i386-netware-gcc`
: The C compiler, GNU CC.

`i386-netware-ld`
: The GNU linker.

`i386-netware-nlmconv`
: Object format converter; used for the last step in building an NLM.

3.2 Compiling for NetWare

Follow these steps to build an NLM:

1. Compile your source files with the '`-c`' option, to build individual object files.
2. Link all your object files with '`prelude.o`' (a Novell run-time initialization file), using the '`-r`' linker option to produce a single relocatable object file.
3. Convert the object file to NLM format, as specified by your header definition file.

For example, these three steps build an NLM called '`hi.nlm`' from a C source file '`hi.c`', the Novell-supplied '`prelude.o`', and an NLM header file '`hi.def`'.

```
$ i386-netware-gcc -c hi.c
                        The '-c' option specifies '.o' output files.

$ i386-netware-gcc -r -o hi.O prelude.o hi.o
                        Use '-r' for relocatable output, '-o' to name the
                        output file.

$ i386-netware-nlmconv -T hi.def hi.O hi.nlm
                        Use '-T' to indicate Novell header definition file;
                        remaining arguments are input file, NLM output file.
```

3.3 Compiler and Linker Options for NetWare

The GNU compiler has a variety of options to cover many needs. The NetWare development environment is based on the Unix System V release 4 environment, and the same command-line options are available as for SVr4. For information on all the GCC command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

GNU CC also has specific options for each hardware architecture. See section "Intel 386 Options" in *Using GNU CC*, for example.

The essential GNU linker options for NLM development are those shown in the example: '`-r`' to make a single relocatable output file, and '`-o` *outfile*' to name the output file.

You can use the GNU CC command-line option '`-Wl`' to combine the compilation and link into a single call to `i386-netware-gcc`:

`-Wl,`*option*

Pass *option* as an option to the linker. If *option* contains commas, it is split into multiple options at the commas.

For example, you can build the '`hi.O`' intermediate file of the previous example with a single call to GNU CC like this. You still need to call `i386-netware-nlmconv`:

```
$ i386-netware-gcc -Wl,-r,-o,hi.O prelude.o hi.c
$ i386-netware-nlmconv -T hi.def hi.O hi.nlm
```

3.4 Making an NLM

Once you've linked a relocatable object file, use the utility `i386-netware-nlmconv` to turn it into an NLM. This utility is very similar to Novell's NLMLINK program (which is used for building NLMs from a DOS development environment). Similarly, the header file '*filename*`.def`' that you must supply uses the same syntax as the header file for NLMLINK, and it recognizes most of the same directives.

Here are the main options and arguments for `nlmconv`.

```
nlmconv [ -T headerfile | --header-file=headerfile ]
        [ infile ]  [ outfile ]
```

See section "nlmconv" in *The GNU Binary Utilities*, for information on a few additional, rarely used, options.

`-T` *headerfile*

`--header-file=`*headerfile*

Reads *headerfile* for NLM header information.

`infile` Name of the single, relocatable, object file to be converted to an NLM.

`outfile` Name for the Netware Loadable Module produced by `nlmconv`.

3.4.1 Differences from DOS development tools

If you have been using `NLMLINK` to develop NLM code, bear these differences in mind when you use `nlmconv`:

Input and output files on command-line
: You can name one input file, and the output file (in that order), on the `nlmconv` command line instead of in the '`filename.def`' file. (You can also use the `OUTPUT` and `INPUT` directives in the definitions file, just as with `NLMLINK`. The `INPUT` directive allows you to list more than one input file, and lets `nlmconv` call the linker for you.)

Option for header file.
: Specify the header file on the command line with the option '`-T filename.def`', *not* as '`@filename.def`'.

Linker a separate program.
: `nlmconv` calls the linker if necessary, but it is not itself a linker. This has no direct impact, but it does mean that `nlmconv` ignores some definition-file directives that would be meaningful only to a linker.

Some directives not supported.
: Because of the foregoing differences, `nlmconv` ignores these directives (which are accepted by `NLMLINK`) in the '`filename.def`' file. To make it easier for you to use the same header file, `nlmconv` generates an NLM even if it prints a warning for one of these directives:

```
MAP          CODESTART
FULLMAP      VERBOSE
```

These unsupported directives are all concerned with displaying a link map. If you use the GNU linker explicitly to make a single `nlmconv` input file, you can use the '`-M`' linker option to display the link map on standard output, or the '`-Map mapfile`' linker option to write the link map in a file of your choice. See section "Command Line Options" in *Using ld: the GNU linker*, for more details.

For example, you can use the GNU linker like this to build a single relocatable object file, and display a link map on standard output:

```
$ i386-netware-ld -M -r -o single.o prelude.o objfiles...
```

3.4.2 What goes in the *file*.def header

You can place comments anywhere in your NLM header file: comments start with the character '#' and end at the next new line. An '@' introduces an include file (its contents are used as if they were in the current file).

Otherwise, each line of a *file*.def header file begins with one of the directives summarized below; the rest of the line contains optional arguments, separated by spaces, tabs, or commas (spaces, tabs and commas are interchangeable). You may type these directives in either upper or lower case. See the "Linkers" chapter of *NetWare NLM Development and Tools Overview*, for more details on these directives.

If you need more than one line for an argument list, type one or more blanks or tabs at the beginning of the continuation lines.

CHECK *ckproc*

Run the check procedure *ckproc* on attempts to unload your NLM; return 0 to indicate unloading is safe, any nonzero value otherwise. There is only one check procedure per NLM; if you specify CHECK more than once, only the last specification takes effect.

nlmconv issues a warning if the check procedure you specify is not in the object code.

COPYRIGHT "*msg*"

Display the copyright notice *msg* on the console screen when your NLM loads.

CUSTOM *filename*

Copy arbitrary data from file *filename* into your output NLM. Your code can reach this data through the customDataOffset and customDataSize fields of the NLM header.

You may use CUSTOM at most once in the NLM header definition file.

DATE *mth dy year*

Force a particular date into the version header of your NLM (otherwise nlmconv uses the current date). Note the argument order: *mth* is the month, *dy* the day, *year* the year. Specify all fields as numbers, and specify the year as an absolute date—for example '1993', not '93'.

`DEBUG`
: Generate debugging information in the format used by the NetWare internal debugger.

`DESCRIPTION "`*`txt`*`"`
: A name for your NLM, displayed on the console whenever it is loaded, and in response to some other console commands.

`EXIT` *`exproc`*
: Run the exit procedure *`exproc`* to clean up any resources your NLM allocated before unloading. This procedure runs *after* everything else in your NLM; in particular, it runs after any C++ destructors, and after any procedures you register with the `atexit` library routine. There is only one exit procedure per NLM; if you specify `EXIT` more than once, only the last specification takes effect.

  The default exit procedure is `_Stop`, defined in Novell's '`prelude.o`' file.

`EXPORT` *`sym1 sym2`* `. . .`
`EXPORT . . . (`*`prefix`*`)` *`sym1 sym2`* `. . .`
: Export a list of symbols (separated by spaces, tabs, or commas) for use by other NLMs. Recall that you can continue the argument list on successive lines if you need to, by starting the continuation lines with blanks or tabs.

  You can insert '`(`*`prefix`*`)`' (a string in parentheses) anywhere in the list; any symbols listed after a '`(prefix)`' will be exported with the *`prefix`* string (followed by the character '`@`') at the beginning of the symbol name. Use a '`(`*`prefix`*`)`' string unique to your NLM to distinguish your symbols from those of other NLMs.

  An empty prefix string '`()`' is valid and useful: use it to cancel the effect of an earlier prefix specification. No '`@`' is inserted when the prefix is an empty string.

  You can use `EXPORT` repeatedly in your header file. Each instance adds more symbols to the export list. Prefixes only apply to the `EXPORT` statement where they appear, however; you must repeat the '`(`*`prefix`*`)`' declaration in each `EXPORT` directive where you want to use the same symbol prefix.

`FLAG_ON` *`bits`*
: Turn on the NLM header flags corresponding to the base-two representation of the number *`bits`*. You can construct a value for *`bits`* by adding any combination of the following:

  `1` Same effect as `REENTRANT` directive.

  `2` Same effect as `MULTIPLE`.

4 Same effect as SYNCHRONIZE.

8 Same effect as PSEUDOPREEMPTION.

16 Same effect as OS_DOMAIN.

FLAG_OFF *bits*
: Turn off the NLM header flags corresponding to the base-two representation of the number *bits*. See FLAG_ON to construct a value for *bits*.

HELP *fname*
: Use file *fname* for any help text supplied by this NLM. You can use this indirection to support multiple languages; you should take care to switch the help file in parallel with the message file.

IMPORT *sym1 sym2* . . .
IMPORT . . . (*prefix*) *sym1 sym2* . . .
: The converse of EXPORT; symbols your NLM needs that are exported by other NLMs. The rules for the list of symbols, and for use of '(*prefix*)', are the same as for EXPORT.

INPUT *objfile* . . .
: You can specify one or more input files using this directive from your header definitions file. nlmconv calls the GNU linker to make a single relocatable file for conversion, if you specify more than one *objfile*.

Warning: it is an error to specify an input file on the command line if your header file also contains the INPUT directive.

MESSAGES *fname*
: Use file *fname* for any messages issued by this NLM. You can use this indirection to support multiple languages; take care to switch the message file in parallel with the help file. Novell provides tools to generate message files.

MODULE *nlm1 nlm2* . . .
: Specify NLMs that must load before yours. You can separate the arguments with spaces, tabs, or commas. Recall that you can continue the argument list on successive lines if you need to, by starting the continuation lines with blanks or tabs.

MULTIPLE
: Include this directive to indicate your NLM may be loaded more than once.

OUTPUT *filename*
: You can specify a default name for the output NLM using this directive. If you also specify an output filename on the nlmconv command line, this directive is ignored.

`OS_DOMAIN`
Include this directive to indicate your NLM is to run in the OS domain—that is, with direct access to all NetWare operating system facilities, and without memory protection.

`PSEUDOPREEMPTION`
Include this directive to make NetWare force your NLM to relinquish control periodically, even if your NLM makes no other arrangement to relinquish control.

`REENTRANT`
If you include this directive, when someone reloads your NLM after it is already loaded, the new thread of control uses the NLM that is already in memory.

`SCREENNAME "`*`nm`*`"`
The string *nm* is a title for your NLM's console display.

`SHARELIB` *`libname`*
Load *libname* as a shared NLM.

`STACK` *`sz`*
`STACKSIZE` *`sz`*
Two equivalent directives to set the stack size to *sz*, for processes in your NLM.

`START` *`stproc`*
Run the start procedure *stproc* to initialize your NLM. The default start procedure is `_Prelude`, defined in Novell's '`prelude.o`'. There is only one start procedure per NLM; if you specify `START` more than once, only the last specification takes effect.

The start procedure runs *before* any C++ constructors.

`SYNCHRONIZE`
Include this directive to indicate that the NetWare OS may not load any further NLMs until your NLM calls the system routine `SynchronizeStart`.

`THREADNAME "`*`txt`*`"`
Use *txt* as a prefix to identify threads that belong to this NLM.

`TYPE` *`num`* A number, describing the NLM "module type". Any value for *num* is accepted, but these are the predefined meanings:

0. Any NLM (this is the default if you do not use `TYPE`)
1. LAN driver
2. Disk driver

3. Name-space support module
4. Utility NLM
5. Mirrored Server Link
6. OS NLM
7. Paged high OS NLM
8. Host Adapter Module
9. Custom Device Module

`VERSION` *maj min*

`VERSION` *maj min rev*

Specify a version number (displayed at the NetWare console on load) for this NLM. *maj min* and *rev* are, respectively, the major version number, minor version number, and revision level; you can specify a revision level or not, as you choose. There are no constraints on major version number, but the minor revision number must be betwen `0` and `99`, and the revision (if you specify it at all) must be between `1` and `26`.

`XDCDATA` *fname*

Copy the contents of file *fname* into your NLM, for use with remote procedure call (RPC) extensions to NetWare.

3.5 NetWare Debugging with GDB

To debug an NLM under GDB, run the NLM under the control of the utility NLM `GDBSERVE.NLM`, and use the GDB `target remote` command.[1] Bear in mind that—as for all other NLM development activity—you should not use a production NetWare system for this purpose. In particular, since NetWare uses cooperative multitasking, *everything* on your NetWare server will come to a halt whenever you stop inside the NLM you are debugging.

GDB and `GDBSERVE` communicate via a serial line. (`GDBSERVE` is an NLM version of the `gdbserver` program; see section "Using the `gdbserver` program" in *Debugging with GDB*, for general information.)

On the NetWare server (the debugging target),

you need to have a copy of the program you want to debug, in NLM format (the output of `nlmconv`).

To use `GDBSERVE`, you must tell it how to communicate with GDB; the name of your program; and the arguments for your program. The syntax is:

[1] `target remote` is otherwise used to debug bare-board systems, in which case it requires linking in special *stub* subroutines; with `GDBSERVE`, no such special stubs are needed.

```
server: LOAD GDBSERVE [ NODE=node ] [ PORT=port ]
     [ BAUD=baud ] program [ args ... ]
```

node and *port* are both numbers: the board and port number respectively, for the serial line attached to the machine where you want to run GDB. For example, to debug an NLM called HI.NLM with the argument 'foo.txt' and communicate with GDB over serial port number 2 on board 1:

```
server: LOAD GDBSERVE NODE=1 PORT=2 HI foo.txt
```

GDBSERVE waits passively for the host GDB to communicate with it.

On the host,
you need the relocatable, unstripped copy of your program that you used as the argument to nlmconv; that is, the original linker output file, not the NLM format object file. GDB reads symbols and debugging information from this file. 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. Its argument is a serial device name (usually something like '/dev/ttyb'). For example:

```
$ i386-netware-gdb hi.O
GDB is free software...
(gdb) target remote /dev/ttyb
```

communicates with the server via serial line '/dev/ttyb'.

3.6 Subroutine Libraries

You must load Novell's C library 'CLIB.NLM' to provide the standard C subroutine libraries for your programs. In future releases, the Cygnus C subroutine libraries may be available as an alternative, pending discussions between Cygnus and Novell on low-level interfaces.

3.7 Predefined Preprocessor Macros

GCC defines these preprocessor macros for NetWare:

```
__netware__
i386
```

3.8 Configuring GNU source for NetWare

Your Cygnus Developer's Kit includes precompiled, ready-to-run binaries, with all defaults configured for your development system.

However, you may have occasion to reconfigure or rebuild the source. For example, you may want to rebuild after receiving a bugfix in source-patch form.

You can use the script `configure` to specify many prearranged kinds of variations in the source. See section "Rebuilding From Source" in *Release Notes*, for a discussion of the source configuration process.

To get the configuration supported by Cygnus for NetWare development, use this `configure` option:

```
--target=i386-netware
```

3.9 NetWare Development Documentation

For general information about programming for NetWare, see *NetWare NLM Development and Tools Overview* (Novell Part # 100–001872–001).

This document is available from Novell, Inc.; 122 East 1700 South; Provo, Utah; 84606 U.S.A. You can also telephone Novell at +1 800 NETWARE.

4 Fujitsu SPARClite Development

The Cygnus Developer's Kit supports the SPARClite family as a variant of the support for SPARC. For the compiler in particular, special configuration options allow you to use special software floating-point code (for the Fujitsu MB86930 chip), as well as defaulting command-line options to use special SPARClite features.

4.1 What to Call the Tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The names are constructed by using as a prefix the '`--target`' argument to `configure`. For example, the compiler (called simply `gcc` in native configurations) is called by one of these names for SPARClite cross-development, depending on which configuration you have installed:

Available as preconfigured binaries from Cygnus:

```
sparclitefrwcompat-aout-gcc
sparclitefrwcompat-coff-gcc
```

Alternatives you can build from source:

```
sparclite-aout-gcc
sparclite-coff-gcc
sparclitefrw-aout-gcc
sparclitefrw-coff-gcc
```

See Section 4.2.1 "Setting up GCC for the SPARClite," page 44, for explanations of the alternative SPARClite configurations.

4.2 Compiling for the SPARClite

When you *configure* GCC itself, you can control what register management strategies to use on the SPARClite, and what kind of software floating-point entry points to generate if hardware floating-point is not available.

When you *run* GCC, you can use command-line options to choose whether to take advantage of the extra SPARClite machine instructions, and whether to generate code for hardware or software floating point.

4.2.1 Setting up GCC for the SPARClite

There are three variants of the SPARClite as a CPU name in GCC configurations, reflecting different register management strategies:

`--target=sparclite-*`
: Generate SPARClite code (including use of additional instructions), but use normal SPARC register management.

`--target=sparclitefrw-*`
: Generate code *only* for the flat register model on the SPARClite; for example, the compiler does not use save and restore.

  Code generated with GCC configured for `sparclitefrw` targets will not link with code from the compiler configured for normal register management.

`--target=sparclitefrwcompat-*`
: Generate code for the flat register model, but in a way that is compatible with normal register management.

  Code generated for the `sparclitefrwcompat` targets is slightly less efficient than code for the `sparclitefrw` targets.

  Three corresponding pseudo-CPU names are available for SPARC chips without the addititional SPARClite instructions: '`--target=sparc-*`', '`--target=sparcfrw-*`', and '`--target=sparcfrwcompat-*`'.

When you run `configure`, you should also specify the object code format you need:

`--target=sparclitefrw-aout`

`--target=sparclitefrwcompat-aout`
: For `a.out` object code.

`--target=sparclitefrw-coff`

`--target=sparclitefrwcompat-coff`
: For COFF object code.

4.2.2 SPARC options for architecture and code generation

Some special compiler command-line options are available for SPARClite; in addition, the machine-dependent options already present for SPARC in general continue to be available. Both kinds of options are described in section "SPARC Options" in *Using GNU CC*.

`-msparclite`
: The SPARC configurations of GCC generate code for the common subset of the instruction set: the v7 variant of the SPARC architecture.

  '`-msparclite`' (which is on automatically for any of the SPARClite configurations) gives you SPARClite code. This adds the integer multiply (`smul` and `umul`, just as in SPARC v8), integer divide-step (`divscc`), and scan (`scan`) instructions which exist in SPARClite but not in SPARC v7.

  Using '`-msparclite`' when you run the compiler does *not*, however, give you floating point code that uses the entry points for US Software's '`goFast`' library. The software floating-point entry points depend on how you *configure* the compiler; with the normal SPARC configuration, GCC generates code that uses the conventional GCC soft-floating-point library entry points. To get the US Software entry points, you must configure the compiler for SPARClite as described above.

`-mv8`
: '`-mv8`' gives you SPARC v8 code. The only difference from v7 code is that the compiler emits the integer multiply (`smul` and `umul`) and integer divide (`sdiv` and `udiv`) instructions which exist in SPARC v8 but not in SPARC v7.

`-mf930`
: Generate code specifically intended for the Fujitsu MB86930, a SPARClite chip without an FPU. This option is equivalent to the combination '`-msparclite -mno-fpu`'.

  '`-mf930`' is the default when the compiler is configured specifically for SPARClite.

`-mf934`
: Generate code specifically for the Fujitsu MB86934, a SPARClite chip *with* an FPU. This option is equivalent to '`-msparclite`'.

The following command line options are available for both SPARClite and other SPARC configurations of the compiler. See section "SPARC Options" in *Using GNU CC*.

`-mno-epilogue`
`-mepilogue`

With '`-mepilogue`' (the default), the compiler always emits code for function exit at the end of each function. Any function exit in the middle of the function (such as a return statement in C) will generate a jump to the exit code at the end of the function.

With '`-mno-epilogue`', the compiler tries to emit exit code inline at every function exit.

The Cygnus Support compiler specialists recommend avoiding '`-mno-epilogue`'.

4.2.3 Compiler command-line options for floating point

When you run the compiler, you can specify whether to compile for hardware or software floating point configurations with these GCC command-line options:

`-mfpu`
`-mhard-float`

Generate output containing floating point instructions. This is the default.

`-msoft-float`
`-mno-fpu`

Generate output containing library calls for floating point. The SPARC configurations of '`libgcc`' include a collection of subroutines to implement these library calls.

In particular, the SPARClite GCC configurations generate subroutine calls compatible with the US Software '`goFast.a`' floating point library, giving you the opportunity to use either the '`libgcc`' implementation or the US Software version.

To use the US Software library, simply include '`-lgoFast`' on the GCC command line.

To use the '`libgcc`' version, you need nothing special; GCC links with '`libgcc`' automatically after all other object files and libraries.

4.2.4 Floating point subroutines

Two kinds of floating point subroutines are useful with GCC:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.

   When you indicate that no hardware floating point is available (with either of the GCC options '`-msoft-float`' or '`-mno-fpu`'), the SPARClite configurations of GCC generate calls compatible with the '`goFast`' library, proprietary licensed software available from U.S. Software. If you do not have this library, you can still use software floating point; '`libgcc`', the auxiliary library distributed with GCC, includes compatible—though slower—subroutines.

2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

4.2.5 SPARC options for unfinished features

Avoid these options—although they appear in the SPARC configuration files, their implementation is not yet complete.

```
-mfrw
-mno-frw
```

4.3 Assembling SPARClite code

The GNU assembler, configured for SPARC, recognizes the additional SPARClite machine instructions that GCC can generate.

You can specify the flag '`-Asparclite`' to the GNU assembler (configured for SPARC) to explicitly select this particular SPARC architecture. In any case, however, the SPARC assembler automatically selects the SPARClite architecture whenever it encounters one of the SPARClite-only instructions (`divscc` or `scan`).

4.4 Remote SPARClite Debugging with GDB

You can use the GDB remote serial protocol to communicate with a SPARClite board. You must first link your programs with the "stub" module '`sparc-stub.c`'; this module manages the communication with GDB. See section "The GDB remote serial protocol" in *Debugging with GDB*, for more details.

4.5 SPARClite documentation

See *SPARClite User's Manual* (Fujitsu Microelectronics, Inc. Semiconductor Division, 1993) for full documentation of the SPARClite family, architecture, and instruction set.