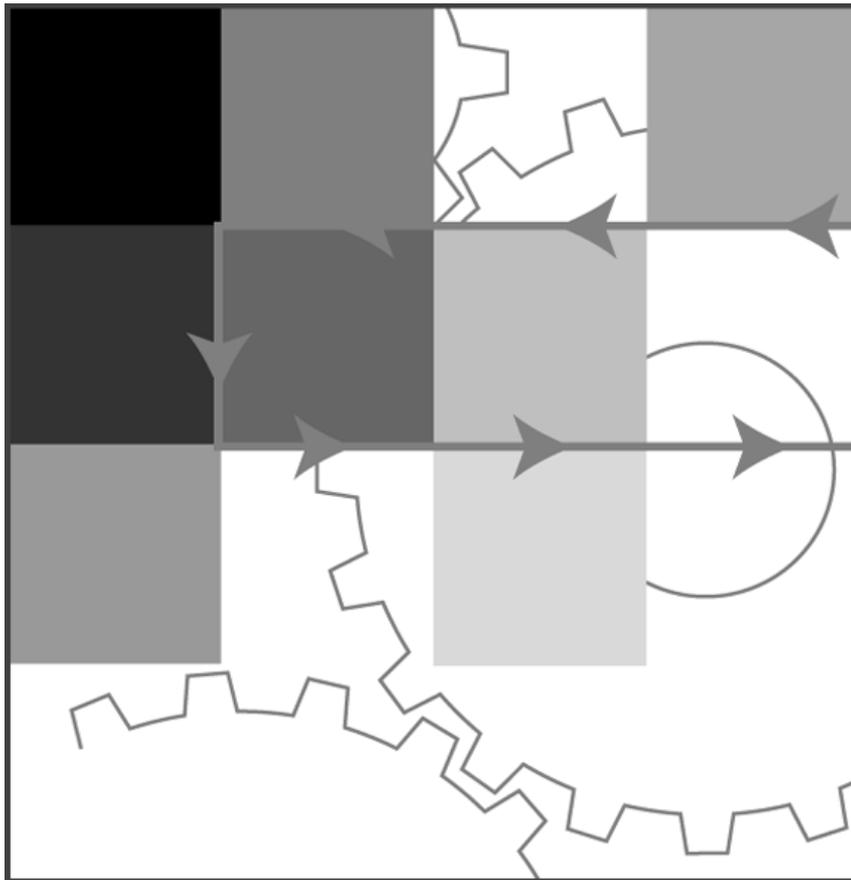


GNUPro® Toolkit

GNUPro Development Tools

- ***Using ld***
- ***Using make***
- ***Using diff & patch***



GNUPro 2001

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

How to Contact Red Hat

Use the following means to contact Red Hat.

Red Hat Corporate Headquarters

2600 Meridian Parkway

Durham, NC 27713 USA

Telephone (toll free): +1 888 REDHAT 1

Telephone (main line): +1 919 547 0012

Telephone (FAX line): +1 919 547 0024

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

Contents

Overview of GNUPro Development Tools	1
--	---

Using ld

Overview of ld, the GNU Linker	5
Invocation of ld, the GNU Linker	7
Using ld Command Line Options.....	8
ld Command Line Options	9
Options Specific to PE Targets	24
ld Environment Variables.....	26
Linker Scripts	29
Basic Linker Script Concepts.....	30
Linker Script Format	31
Simple Linker Script Example	31
Simple Linker Script Commands	32
Setting the Entry Point	32
Commands Dealing with Files.....	33
Commands Dealing with Object File Formats	34
Other Linker Script Commands	34
Assigning Values to Symbols	35
Simple Assignments	35
PROVIDE Keyword.....	36

SECTIONS Command	37
Output Section Description.....	38
Output Section Name.....	38
Output Section Address	39
Input Section Description	39
Input Section Basics.....	39
Input Section Wildcard Patterns	40
Input Section for Common Symbols	41
Input Section and Garbage Collection.....	42
Input Section Example.....	42
Output Section Data.....	42
Output Section Keywords.....	43
Output Section Discarding.....	44
Output Section Attributes	45
Overlay Description.....	47
MEMORY Command	49
PHDRS Command	50
VERSION Command	53
Expressions in Linker Scripts	55
Constants.....	55
Symbol Names.....	56
The Location Counter	56
Operators.....	57
Evaluation	57
The Section of an Expression	58
Builtin Functions.....	58
Implicit Linker Scripts	61
1a Machine Dependent Features	63
1a and the H8/300 Processors	63
1a and Intel 960 Processors.....	64
1a Support for Interworking Between ARM and Thumb Code.....	65
BFD Library	67
How BFD Works (an Outline of BFD).....	68
Information Loss	68
The BFD Canonical Object File Format	69
MRI Compatible Script Files for the GNU Linker	71

Using make

Overview of <code>make</code>, a Program for Recompiling	77
Introduction to Makefiles	79

Makefile Rule's Form	80
A Simple Makefile	81
How make Processes a Makefile	82
Variables Make Makefiles Simpler.....	83
Letting make Deduce the Commands	84
Another Style of Makefile.....	85
Rules for Cleaning the Directory	86
Writing Makefiles	87
What Makefiles Contain	88
What Name to Give Your Makefile	88
Including Other Makefiles	89
The MAKEFILES Variable	90
How Makefiles are Remade	91
Overriding Part of Another Makefile.....	92
Writing Rules	93
Rule Syntax	94
Using Wildcard Characters in File Names.....	95
Pitfalls of Using Wildcards	96
The wildcard Function.....	96
Searching Directories for Dependencies	97
vpath : Search Path for All Dependencies	98
The vpath Directive.....	98
How Directory Searches Work	99
Writing Shell Commands with Directory Search	100
Directory Search and Implicit Rules.....	101
Directory Search for Link Libraries.....	101
Phony Targets.....	101
Rules Without Commands or Dependencies.....	103
Empty Target Files to Record Events	103
Special Built-in Target Names	104
Multiple Targets in a Rule.....	105
Multiple Rules for One Target.....	106
Static Pattern Rules	107
Syntax of Static Pattern Rules	107
Static Pattern Rules Compared to Implicit Rules	108
Double-colon Rules.....	109
Generating Dependencies Automatically.....	109
Writing the Commands in Rules	113
Command Echoing.....	114
Command Execution.....	114
Parallel Execution	116
Errors in Commands	117

Interrupting or Killing the <code>make</code> Tool.....	118
Recursive Use of the <code>make</code> Tool.....	119
How the <code>MAKE</code> Variable Works	119
Communicating Variables to a Sub- <code>make</code> Utility.....	120
Communicating Options to a Sub- <code>make</code> Utility	122
The <code>--print-directory</code> Option.....	123
Defining Canned Command Sequences.....	124
Using Empty Commands	125
How to Use Variables	127
Basics of Variable References	128
The Two Flavors of Variables	129
Substitution References	131
Computed Variable Names	132
How Variables Get Their Values	134
Setting Variables	135
Appending More Text to Variables	135
The <code>override</code> Directive.....	137
Defining Variables Verbatim	137
Variables from the Environment.....	138
Target-specific Variable Values.....	139
Pattern-specific Variable Values.....	140
Conditional Parts of Makefiles	141
Syntax of Conditionals.....	143
Conditionals That Test Flags	145
Functions for Transforming Text	147
Function Call Syntax.....	148
Functions for String Substitution and Analysis	148
Functions for File Names.....	151
The <code>foreach</code> Function.....	153
The <code>origin</code> Function	155
The <code>shell</code> Function	156
How to Run the <code>make</code> Tool	159
Arguments to Specify the Goals	160
Instead of Executing the Commands	162
Avoiding Recompilation of Some Files.....	164
Overriding Variables.....	164
Testing the Compilation of a Program	165
Summary of <code>make</code> Options	167
Implicit Rules	173
Using Implicit Rules	174
Catalogue of Implicit Rules	175
Variables Used by Implicit Rules	179

Chains of Implicit Rules.....	181
Defining and Redefining Pattern Rules.....	182
Fundamentals of Pattern Rules	182
Pattern Rule Examples.....	183
Automatic Variables	184
How Patterns Match.....	187
Match-anything Pattern Rules	187
Canceling Implicit Rules	188
Defining Last-resort Default Rules	188
Old-fashioned Suffix Rules.....	189
Implicit Rule Search Algorithm	191
Using <code>make</code> to Update Archive Files	193
Archive Members as Targets	194
Implicit Rule for Archive Member Targets	194
Updating Archive Symbol Directories	195
Dangers When Using Archives.....	195
Suffix Rules for Archive Files	196
Summary of the Features for the GNU <code>make</code> utility	197
GNU <code>make</code>'s Incompatibilities and Missing Features.....	201
Problems and Bugs with <code>make</code> Tools.....	203
Makefile Conventions.....	205
General Conventions for Makefiles	205
Utilities in Makefiles.....	207
Standard Targets for Users.....	207
Variables for Specifying Commands	211
Variables for Installation Directories	212
Install Command Categories.....	216
GNU <code>make</code> Quick Reference.....	219
Directives that <code>make</code> Uses.....	220
Text Manipulation Functions	221
Automatic Variables that <code>make</code> Uses	222
Variables that <code>make</code> Uses.....	223
Error Messages that <code>make</code> Generates.....	224
Complex Makefile Example	227

Using `diff` & `patch`

Overview of <code>diff</code> & <code>patch</code>, the Compare & Merge Tools	235
What Comparison Means	237
Hunks	238
Suppressing Differences in Blank and Tab Spacing.....	239

Suppressing Differences in Blank Lines	239
Suppressing Case Differences	240
Suppressing Lines Matching a Regular Expression	240
Summarizing Which Files Differ	240
Binary Files and Forcing Text Comparisons	241
diff Output Formats	243
Two Sample Input Files	244
Showing Differences Without Context	244
Detailed Description of Normal Format	245
An Example of Normal Format	245
Showing Differences in Their Context	246
Context Format	246
Unified Format	248
Showing Sections In Which There Are Differences	249
Showing Alternate File Names	250
Showing Differences Side by Side	251
Controlling Side by Side Format	252
An Example of Side by Side Format	252
Making Edit Scripts	252
ed Scripts	252
Detailed Description of ed Format	253
Example ed Script	254
Forward ed Scripts	254
RCS Scripts	254
Merging Files with If-then-else	255
Line Group Formats	255
Line Formats	258
Detailed Description of If-then-else Format	259
An Example of If-then-else Format	260
Comparing Directories	261
Making diff Output Prettier	263
Preserving Tabstop Alignment	263
Paginating diff Output	264
diff Performance Tradeoffs	265
Comparing Three Files	267
A Third Sample Input File	268
Detailed Description of diff3 Normal Format	268
diff3 Hunks	269
An Example of diff3 Normal Format	269
Merging from a Common Ancestor	271
Selecting Which Changes to Incorporate	272
Marking Conflicts	273

Generating the Merged Output Directly	274
How <code>diff3</code> Merges Incomplete Lines	274
Saving the Changed File	275
sdiff Interactive Merging	277
Specifying <code>diff</code> Options to the <code>sdiff</code> Utility.....	278
Merge Commands	278
Merging with the <code>patch</code> Utility	281
Selecting the <code>patch</code> Input Format	282
Applying Imperfect Patches	282
Applying Patches with Changed White Space	282
Applying Reversed Patches	283
Helping <code>patch</code> Find Inexact Matches	283
Removing Empty Files.....	284
Multiple Patches in a File.....	284
Messages and Questions from the <code>patch</code> Utility	285
Tips for Making Distributions with Patches	287
Invoking the <code>cmp</code> Utility	289
<code>cmp</code> Options	289
Invoking the <code>diff</code> Utility	291
<code>diff</code> Options	292
Invoking the <code>diff3</code> Utility	299
<code>diff3</code> Options	299
Invoking the <code>patch</code> Utility	303
Applying Patches in Other Directories	304
Backup File Names	304
Naming Reject Files.....	305
<code>patch</code> Options	306
Invoking the <code>sdiff</code> Utility	311
<code>sdiff</code> Options	312
Incomplete Lines	315
Future Projects for <code>diff</code> and <code>patch</code> Utilities	317
Suggested Projects for Improving GNU <code>diff</code> and <code>patch</code> Utilities	318
Handling Changes to the Directory Structure	318
Files That Are Neither Directories Nor Regular Files	318
File Names That Contain Unusual Characters	319
Arbitrary Limits	319
Handling Files That Do Not Fit in Memory.....	319
Ignoring Certain Changes	319
Reporting Bugs.....	320
Index	321

Overview of GNUPro Development Tools

The following documentation comprises the contents of the *GNUPro Development Tools*.

- “Using ld”
(for contents, see “Overview of ld, the GNU Linker” on page 5)
- “Using make”
(for contents, see “Overview of make, a Program for Recompiling” on page 77)
- “Using diff & patch”
(for contents, see “Overview of diff & patch, the Compare & Merge Tools” on page 235)

Using 1d

Copyright © 1991-2000 Free Software Foundation, Inc.

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

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the “GNU General Public License” 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 documentation into another language, under the above conditions for modified versions, except that the documentation 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. For more details, see “General Licenses and Terms for Using GNUPro Toolkit” in *Getting Started Guide*.

Free Software Foundation

59 Temple Place / Suite 330
Boston, MA 02111-1307 USA

ISBN: 1-882114-66-3

1

Overview of `ld`, the GNU Linker

Linkers allow you to build your programs from modules rather than as huge source files. The GNU linker, `ld`, combines object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run `ld`. The following documentation discusses the basics of using the GNU linker.

- “Invocation of `ld`, the GNU Linker” on page 7
- “Linker Scripts” on page 29
- “`ld` Machine Dependent Features” on page 63
- “BFD Library” on page 67
- “MRI Compatible Script Files for the GNU Linker” on page 71

`ld` accepts Linker Command Language files written in a superset of AT&T’s *Link Editor Command Language* syntax^{*}, providing explicit and total control over the

^{*} A standard from the System V UNIX convention, enabling the linker to have one source from which the compiler or assembler creates object files containing the binary code and data for an executable.

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, ELF, COFF or `a.out`. The linker is capable of performing partial links and, for certain executable formats, it can also produce shared libraries or Dynamic Link Libraries (DLLs). Different formats may be linked together to produce any available kind of object file. See “BFD Library” on page 67 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 of `ld`, the GNU Linker

`ld`, the GNU linker, 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. In most circumstances, GCC is capable of running `ld` for you, passing the correct command line options; however, it may be better or easier for you to invoke `ld` directly, in order to override GCC's default choices. The following documentation discusses using the GNU linker with such choices.

- “Using `ld` Command Line Options” on page 8
- “`ld` Command Line Options” on page 9
- “`ld` Environment Variables” on page 26

See also “Linker Scripts” on page 29.

Using `ld` Command Line Options

The linker supports many command line options; 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`, you use the following example's input.

```
ld -o output /lib/crt0.o hello.o -lc
```

This tells `ld` to produce a file called `output` as the result of linking the `/lib/crt0.o` file with `hello.o` and the `libc.a` library, which will come from the standard search directories; see the discussion of search directories with the `-L` option on page 12.

Some of the command line options to `ld` may be specified at any point in the command line sequence. However, options which refer to files, such as `-l` or `-T`, cause the file to be read at the point at which the option appears in the sequence, relative to the object files and other file options. Repeating non-file 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. Options which may be meaningfully specified more than once are noted in the following discussions.

Non-option arguments are object files which are to be linked together. They may follow, precede, or be mixed in with command line options, except that an object file 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 a `No input files` message. 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. Specifying a script in this way should only be used to augment the main linker script; if you want to use a command that logically can only appear once, such as the `SECTIONS` or `MEMORY` command, replace the default linker script using the `-T` option; see the documentation with “Linker Scripts” on page 29 for more discussion.

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.

If the linker is being invoked indirectly, using a compiler driver (for example, with the `gcc` command), then all the linker command line options should be prefixed by `-Wl,` (or whatever is appropriate for the particular compiler driver). The following example shows usage.

```
gcc -Wl,--startgroup foo.o bar.o -Wl,--endgroup
```

If you don't specify the `-Wl,` flag, the compiler driver program may silently drop the linker options, resulting in a bad link.

ld Command Line Options

The following options are for using the GNU linker.

`-akeyword`

This option is supported for HP/UX compatibility. The keyword argument must be one of the `archive`, `shared`, or `default` strings. `-archive` is functionally equivalent to `-Bstatic`, and the other two keywords are functionally equivalent to `-Bdynamic`. This option may be used any number of times.

`-Architecture`

`--architecture=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 “ld and Intel 960 Processors” on page 64 for details. Future releases of `ld` may support similar functionality for other architecture families.

`-b input-format`

`--format=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 configured 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 a `objdump -i` call.) `-format input-format` has the same effect, as does the script command `TARGET`. See “BFD Library” on page 67.

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 “ld Environment Variables” on page 26. You can also define the input format from a script, using the command `TARGET`.
- `-c MRI-commandfile`
`--mri-script=MRI-commandfile`
- For compatibility with linkers produced by MRI, `ld` accepts script files written in an alternate, restricted command language; see “MRI Compatible Script Files for the GNU Linker” on page 71. 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-commandfile` 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.
- `-e entry`
`--entry=entry`
- Use `entry` as the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named `entry`, the linker will try to parse `entry` as a number, using that as the entry address (the number will be interpreted in base 10; you may use a leading `0x` for base 16, or a leading `0` for base 8). See “Setting the Entry Point” on page 32 for a discussion of defaults and other ways of specifying the entry point.
- `-E`
`--export-dynamic`
- When creating a dynamically linked executable, add all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols which are visible from dynamic objects at run time.
- If you do not use this option, the dynamic symbol table will normally contain only those symbols which are referenced by some dynamic object mentioned in the link.
- If you use `dlopen` to load a dynamic object which needs to refer back to the symbols defined by the program, rather than some other dynamic object, then you will probably need to use this option when linking the program itself.
- `-EB`
- Link big-endian objects. This affects the default output format.
- `-EL`
- Link little-endian objects. This affects the default output format.
- `-f name`
`--auxiliary name`
- When creating an ELF shared object, set the internal `DT_AUXILIARY` field to the

specified name. This tells the dynamic linker that the symbol table of the shared object should be used as an auxiliary filter on the symbol table of the shared object *name*.

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the `DT_AUXILIARY` field. If the dynamic linker resolves any symbols from the filter object, it will first check whether there is a definition in the shared object *name*. If there is one, it will be used instead of the definition in the filter object. The shared object *name* need not exist. Thus the shared object *name* may be used to provide an alternative implementation of certain functions, perhaps for debugging or for machine specific performance.

This option may be specified more than once. The `DT_AUXILIARY` entries will be created in the order in which they appear on the command line.

`-F name`

`--filter name`

When creating an ELF shared object, set the internal `DT_FILTER` field to the specified name. This tells the dynamic linker that the symbol table of the shared object should be used as a filter on the symbol table of the shared object *name*.

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the `DT_FILTER` field. The dynamic linker will resolve any symbols, according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object *name*. Thus the filter object *name* may be used to select a subset of the symbols provided by the object *name*.

Some older linkers used the `-F` option throughout a compilation toolchain for specifying object-file format for both input and output object files. The GNU linker uses other mechanisms for this purpose: the `-b`, `--format`, `--oformat` options, the `TARGET` command in linker scripts, and the `GNUTARGET` environment variable. The GNU linker will ignore the `-F` option when not creating an ELF shared object.

`-g`

Ignored. Provided for compatibility with other tools.

`-Gvalue`

`--gpsize=value`

Set the maximum size of objects to be optimized using the `GP` register to *size*.

This is only meaningful for object file formats such as MIPS ECOFF which supports putting large and small objects into different sections. Ignored for other object file formats.

`-hname`

`-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, so that, then, when the executable is run, the dynamic linker will

attempt to load the shared object specified by the `DT_SONAME` field rather than using the file name given to the linker.

`-i`

Perform an incremental link (same as option `-r`).

`-larchive`

`--library=archive`

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 library *archive.a* for every *archive* specified.

On systems which support shared libraries, `ld` may also search for libraries with extensions other than `.a`. Specifically, on ELF and SunOS systems, `ld` will search a directory for a library with an extension of `.so` before searching for one with an extension of `.a`. By convention, a `.so` extension indicates a shared library.

The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol which was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.

See the `-(archives-)` option on page 15 for a way to force the linker to search archives multiple times.

You may list the same archive multiple times on the command line.

This type of archive searching is standard for UNIX linkers. However, if you are using `ld` on AIX, note that it is different from the behaviour of the AIX linker.

`--library-path=dir`

`-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 directories are searched in the order in which they are specified on the command line. Directories specified on the command line are searched before the default directories. All `-L` options apply to all `-l` options, regardless of the order in which the options appear. 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 “`ld` Environment Variables” on page 26.

The paths can also be specified in a link script with the `SEARCH_DIR` command. Directories specified this way are searched at the point in which the linker script appears in the command line.

`-memulation`

Emulate the *emulation* linker. You can list the available emulations with the `--verbose` or `-v` options. The default depends on the configuration of `ld`.

If the `-m` option is not used, the emulation is taken from the `LDEMULATION`

environment variable, if that is defined.

Otherwise, the default emulation depends upon how the linker was configured.

-M

--print-map

Print a link map to the standard output; a link map provides information about the link, including the following information.

- Where object files and symbols are mapped into memory.
- How common symbol files are allocated.
- All archive members included in the link, with a mention of the symbol which caused the archive member to be brought in.

-n

--nmagic

Turn off page alignment of sections, and mark the output as `NMAGIC` if possible.

-N

--omagic

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`.

-o *output*

--output=*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 `OUTPUT` script command can also specify the output file name.

-O *level*

If *level* is a numeric values greater than zero, `ld` optimizes the output. This might take significantly longer and therefore probably should only be enabled for the final binary.

-q

--emit-relocs

Preserve the relocation sections in the final output.

-r

--relocatable

Generate relocatable output; that is, 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`.

-R *filename*

--just-symbols=*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. You may use this option more than once.

For compatibility with other ELF linkers, if the `-R` option is followed by a directory name, rather than a file name, it is treated as the `-rpath` option.

`-s`

`--strip-all`

Omit all symbol information from the output file.

`-S`

`--strip-debug`

Omit debugger symbol information (but not all symbols) from the output file.

`-t`

`--trace`

Print the names of the input files as ld processes them.

`-T scriptfile`

`--script=scriptfile`

Use *scriptfile* as the linker script. This script replaces ld's default link script (rather than adding to it), so *scriptfile* must specify everything necessary to describe the output file. You must use this option if you want to use some command that can appear only once in a linker script, such as the `SECTIONS` or `MEMORY` command (see "Linker Scripts" on page 29). If *scriptfile* does not exist, ld looks for it in the directories specified by any preceding `-L` options. Multiple `-T` options accumulate.

`-u symbol`

`--undefined=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. This option is equivalent to the `EXTERN` linker script command.

`-Ur`

For anything other than C++ programs, this option is equivalent to `-r`; it generates relocatable output, meaning that the output file 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.

`-v`

`--version`

`-V`

Display the version number for ld. The `-v` option also lists the supported emulations.

`-x`

`--discard-all`

Delete all local symbols.

- X
- discard-locals
Delete all temporary local symbols. For most targets, this is all local symbols whose names begin with `L`.
- y *symbol*
- trace-symbol=*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 you are not certain of its origins.
- Y *path*
Add *path* to the default library search path. This option exists for Solaris compatibility.
- (*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. Use it only when there are unavoidable circular references between two or more archives.
- assert *keyword*
This option is ignored for Solaris compatibility.
- Bdynamic
- dy
- call-shared
Link against dynamic libraries. This is only meaningful on platforms for which shared libraries are supported. This option is normally the default on such platforms. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the commandline: it affects library searching for `-l` options that follow it.
- Bstatic
- dn
- non_shared
- static
Do not link against shared libraries. This is only meaningful on platforms for which shared libraries are supported. This option is normally the default on such platforms. The different variants of this option are for compatibility with various

systems. You may use this option multiple times on the commandline: it affects library searching for `-l` options that follow it.

`-Bsymbolic`

When creating a shared library, bind references to global symbols to the definition within the shared library, if any. Normally, it is possible for a program linked against a shared library to override the definition within the shared library. This option is only meaningful on ELF platforms that support shared libraries.

`--check-sections`

`--no-check-sections`

`--no-check-sections` asks the linker *not* to check section addresses after they have been assigned to see if there any overlaps. Normally the linker will perform this check; if it finds any overlaps it will produce suitable error messages. The linker does know about and does make allowances for sections in overlays. The default behaviour can be restored by using the command line switch,

`--check-sections`.

`--cref`

Output a cross reference table. If a linker map file is being generated, the cross reference table is printed to the map file. Otherwise, it is printed on the standard output.

The format of the table is intentionally simple, so that it may be easily processed by a script if necessary. The symbols are printed out, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

`--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 “Assigning Values to Symbols” on page 35).

IMPORTANT! There should be no white space between *symbol*, the equals sign (`=`), and *expression*.

`--demangle`

`--no-demangle`

These options control whether to demangle symbol names in error messages and other output. When the linker is told to demangle, it tries to present symbol names in a readable fashion: it strips leading underscores if they are used by the object file format, and converts C++ mangled symbol names into user readable names.

The linker will demangle by default unless the environment variable, `COLLECT_NO_DEMANGLE`, is set. These options may be used to override the default.

`--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 the `ld/testsuite/ld-empic` directory for details.

`--errors-to-file file`

Send error messages to *file* instead of printing them on the standard error output.

`-fini name`

When creating an ELF executable or shared object, call *name* when the executable or shared object is unloaded, by setting `DT_FINI` to the address of the function. By default, the linker uses `_fini` as the function to call.

`--force-exe-suffix`

Make sure that an output file has a `.exe` suffix.

If a successfully built fully linked output file does not have a `.exe` or `.dll` suffix, this option forces the linker to copy the output file to one of the same name with a `.exe` suffix. This option is useful when using unmodified UNIX makefiles on a Microsoft Windows host, since some versions of Windows won't run an image unless it ends in a `.exe` suffix.

`--no-gc-sections`

`--gc-sections`

`--gc-sections` enables garbage collection of unused input sections; it is ignored on targets that do not support this option, and is not compatible with `-r`, nor should it be used with dynamic linking. The default behaviour (of not performing this garbage collection) can be restored by specifying `--no-gc-sections`.

`-help`

Print a summary of the command line options on the standard output and exit.

`-init name`

When creating an ELF executable or shared object, call *name* when the executable or shared object is loaded, by setting `DT_INIT` to the address of the function. By default, the linker uses `_init` as the function to call.

`-Map mapfile`

Print a link map to the file *mapfile*. See the description for `-M` and `--print-map` on page 13.

`--no-keep-memory`

`ld` normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option 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.
- `--no-undefined`
Normally when creating a non-symbolic shared library, undefined symbols are allowed and left to be resolved by the runtime loader. This option disallows such undefined symbols.
- `--no-warn-mismatch`
Normally `ld` will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells `ld` that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures that the linker errors are inappropriate.
- `--no-whole-archive`
Turn off the effect of the `--whole-archive` option for subsequent archive files.
- `--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.
- `-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 use this option, as `ld` should be configured to produce, as a default output format, the format most common 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 “BFD Library” on page 67.
- `-qmagic`
This option is ignored for Linux compatibility.
- `-Qy`
This option is ignored for SVR4 compatibility.
- `--relax`
An option with machine dependent effects. Currently this option is only supported on the H8/300 and the Intel 960. See “`ld` and the H8/300 Processors” on page 63 and “`ld` and Intel 960 Processors” on page 64.
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 some platforms, these link time global optimizations may make symbolic debugging of the resulting executable impossible (for instance, for the Matsushita MN10200 and MN10300 processors).

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 runtime 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 used 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.

The `-rpath` option is also used when locating shared objects which are needed by shared objects explicitly included in the link; see the description of the `-rpath-link` option. If `-rpath` is not used when linking an ELF executable, the contents of the environment variable `LD_RUN_PATH` will be used if it is defined.

The `-rpath` option may also be used on SunOS. By default, on SunOS, the linker will form a runtime search path out of all the `-L` options it is given. If a `-rpath` option is used, the runtime search path will be formed exclusively using the `-rpath` options, ignoring the `-L` options. This can be useful when using `gcc`, which adds many `-L` options which may be on NFS mounted filesystems.

For compatibility with other ELF linkers, if the `-R` option is followed by a directory name, rather than a file name, it is treated as the `-rpath` option.

`-rpath-link DIR`

When using ELF or SunOS, one shared library may require another. This happens when an `ld -shared` link includes a shared library as one of the input files. When the linker encounters such a dependency when doing a non-shared, non-relocateable link, it will automatically try to locate the required shared library and include it in the link, if it is not included explicitly. In such a case, the `-rpath-link` option specifies the first set of directories to search. The `-rpath-link` option may specify a sequence of directory names either by specifying a list of names separated by colons, or by appearing multiple times. The linker uses the following search paths to locate required shared libraries.

- Any directories specified by `-rpath-link` options.
- Any directories specified by `-rpath` options. `-rpath` and `-rpath-link` differ in that directories specified by `-rpath` are included in the executable to use at runtime, while the `-rpath-link` is only effective at link time.
- On an ELF system, if the `-rpath` and `rpath-link` options were not used, search the contents of the environment variable, `LD_RUN_PATH`.

- On SunOS, if the `-rpath` option was not used, search any directories specified using `-L` options.
- For a native linker, the contents of the environment variable `LD_LIBRARY_PATH`.
- The default directories, normally `/lib` and `/usr/lib`.

If the required shared library is not found, the linker will issue a warning and continue with the link.

`--section-start-name=address`

Sets the start address of a section called *name* to be *address*.

`-shared`

`-Bshareable`

Create a shared library. This is currently only supported on ELF, XCOFF and SunOS platforms. On SunOS, the linker will automatically create a shared library if the `-e` option is not used and there are undefined symbols in the link.

`--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-file`

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

`-split-by-reloc count`

Tries to create 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.

`-stats`

Compute and display statistics about the operation of the linker, such as execution time and memory usage.

`--task-link`

Perform task level linking. This is similar to performing a relocatable link except that defined global symbols are also converted into static symbols as well.

`-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, reducing the size of an output file with full debugging information by over 30%. 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.

`-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.

`--dll-verbose`

`--verbose`

Display the version number for `ld` and list the linker emulations supported.

Display which input files can and cannot be opened. Display the linker script if using a default builtin script.

`--version-exports-entry symbol_name`

Records *symbol_name* as the version symbol to be applied to the symbols listed for export in the object file's `.export` section.

`--version-script=version-scriptfile`

Specify the name of a version script to the linker. This is typically used when creating shared libraries to specify additional information about the version heirarchy for the library being created. This option is only meaningful on ELF platforms which support shared libraries. See the documentation that starts with “VERSION Command” on page 53.

`--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 the following 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.

- 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
```

- 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
```

- 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
```

- 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
```

- Merging a common symbol with a previous smaller common symbol. The following 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-constructors`

Warn if any global constructors are used. This is only useful for a few object file formats. For formats like COFF or ELF, the linker can not detect the use of global constructors.

`--warn-multiple-gp`

Warn if multiple global pointer values are required in the output file. This is only meaningful for certain processors, such as the Alpha. Specifically, some processors put large-valued constants in a special section. A special register (the global pointer) points into the middle of this section, so that constants can be loaded efficiently using a base-register relative addressing mode. Since the offset in base-register relative mode is fixed and relatively small (that is, 16 bits), this limits the maximum size of the constant pool. Thus, in large programs, it is often necessary to use multiple global pointer values in order to be able to address all

possible constants. This option causes a warning to be issued whenever this case occurs.

`-warn-once`

Warn once for each undefined symbol, rather than once per module referring to it.

`--warn-section-align`

Warn if the address of an output section is changed because of alignment.

Typically, the alignment will be set by an input section. The address will only be changed if it not explicitly specified; that is, if the `SECTIONS` command does not specify a start address for the section (see the documentation that starts with the discussions for “`SECTIONS` Command” on page 37).

`--whole-archive`

For each archive mentioned on the command line after the `--whole-archive` option, 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. This option may be used more than once.

`--wrap symbol`

Use a wrapper function for *symbol*. Any undefined reference to *symbol* will be resolved to `__wrap_symbol`. Any undefined reference to `__real_symbol` will be resolved to *symbol*.

This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_symbol`. If it wishes to call the system function, it should call `__real_symbol`.

Here is a trivial example:

```
void *
__wrap_malloc (int c)
{
    printf ("malloc called with %ld\n", c);
    return __real_malloc (c);
}
```

If you link other code with this file, using `--wrap malloc`, then all calls to `malloc` will call the `__wrap_malloc` function instead. The call to `__real_malloc` in `__wrap_malloc` will call the real `malloc` function.

You may wish to provide a `__real_malloc` function as well, so that links without the `--wrap` option will succeed. If you do this, you should not put the definition of `__real_malloc` in the same file as `__wrap_malloc`; if you do, the assembler may resolve the call before the linker has a chance to wrap it to `malloc`.

The following command line options are specific to ELF format.

`--enable-new-dtags`

Enables the creation of a new format of dynamic tags.

`--disable-new-dtags`

Restores the default, old style dynamic tags.

- z `initfirst`
Mark an object file as the first to be initialized at run-time.
- z `interpose`
Mark to interpose all Dynamic Shared Object (DSO) files, in order to provide a way to build a piece of program code in a special format for loading it at run-time into the address space of an executable program.
- z `lodfltr`
Mark an object as requiring immediate processing.
- z `nodefaultlib`
Mark an object to avoid using default search libraries.
- z `nodelete`
Mark an object as not deletable at run-time.
- z `nodlopen`
Mark an object as not available to `dlopen()`.
- z `nodump`
Mark an object as not available to `dldump()`.
- z `now`
Mark an object as requiring non-lazy run-time binding.
- z `origin`
Mark an object as requiring immediate `$ORIGIN` processing at run-time.

Options Specific to PE Targets

The PE linker supports the `-shared` option, which causes the output to be a dynamically linked library (DLL) instead of a normal executable. You should name the output `*.dll` when you use this option. In addition, the linker fully supports the standard `*.def` files, which may be specified on the linker command line like an object file (in fact, it should precede archives it exports symbols from, to ensure that they get linked in, just like a normal object file).

In addition to the options common to all targets, the PE linker support additional command line options that are specific to the PE target. Options that take values may be separated from their values by either a space or an equals sign.

- `--add-stdcall-alias`
If given, symbols with a `stdcall` suffix (`@nn`) will be exported as-is and also with the suffix stripped.
- `--base-file file`
Use `file` as the name of a file in which to save the base addresses of all the relocations needed for generating DLLs with `dlltool`.
- `--compat-implib`
Create a backwards compatible import library and create `__imp_symbol` symbols as well.
- `--dll`
Create a DLL instead of a regular executable. You may also use `-shared` or specify a `LIBRARY` in a given `.def` file.

--enable-auto-image-base

--disable-auto-image-base

With `--enable-auto-image-base`, automatically choose an image base for DLLs, unless one is provided by your source file. With

`--disable-auto-image-base`, restore the default behavior.

--enable-stdcall-fixup

--disable-stdcall-fixup

If the linker finds a symbol that it cannot resolve, it will attempt to do *fuzzy linking* by looking for another defined symbol that differs only in the format of the symbol name (`cdecl` vs `stdcall`) and will resolve that symbol by linking to the match. For example, the undefined `_foo` symbol might be linked to the `_foo@12` function, or the undefined symbol, `_bar@16`, might be linked to the `_bar` function. When the linker does this, it prints a warning, since it normally should have failed to link, but sometimes import libraries generated from third-party DLLs may need this feature to be usable. If you specify `--enable-stdcall-fixup`, this feature is fully enabled and warnings are not printed. If you specify

`--disable-stdcall-fixup`, this feature is disabled and such mismatches are considered to be errors.

--export-all-symbols

If given, all global symbols in the objects used to build a DLL will be exported by the DLL. This is the default if there otherwise wouldn't be any exported symbols. When symbols are explicitly exported using DEF files or implicitly exported using function attributes, the default is to not export anything else unless this option is given. The `DllMain@12`, `DllEntryPoint@0`, and `impure_ptr` symbols will not be automatically exported.

--exclude-symbols *symbol, symbol, . . .*

Specifies a list of symbols (*symbol*) which should not be automatically exported. The symbol names may be delimited by commas or colons.

--file-alignment

Specify the file alignment. Sections in the file will always begin at file offsets which are multiples of this number. This defaults to 512.

--heap *reserve*

--heap *reserve, commit*

Specify the amount of memory to reserve (and, optionally, commit) to be used as heap for this program. The default is 1Mb reserved, 4K committed.

--image-base *value*

Use *value* as the base address of your program or DLL. This is the lowest memory location that will be used when your program or DLL is loaded. To reduce the need to relocate and improve performance of your DLLs, each should have a unique base address and not overlap any other DLLs. The default is 0x400000 for executables, and 0x1000000 for DLLs.

- `--kill-at`
If given, the `stdcall` suffixes (`@nn`) will be stripped from symbols before they are exported.
- `--major-image-version value`
Sets the major number of the image version. *value* defaults to 1.
- `--major-os-version value`
Sets the major number of the operating system version. *value* defaults to 4.
- `--major-subsystem-version value`
Sets the major number of the subsystem version. *value* defaults to 4.
- `--minor-image-version value`
Sets the minor number of the image version. *value* defaults to 0.
- `--minor-os-version value`
Sets the minor number of the operating system version, os version. *value* defaults to 0.
- `--minor-subsystem-version value`
Sets the minor number of the subsystem version. *value* defaults to 0.
- `--out-implib file`
Generate an import library.
- `--output-def file`
The linker will create the file, *file*, which will contain a DEF file corresponding to the DLL the linker is generating. This DEF file (which should have a `.def` extension) may be used to create an import library with `dlltool` or may be used as a reference to automatically or implicitly exported symbols.
- `--section-alignment`
Sets the section alignment. Sections in memory will always begin at addresses which are a multiple of this number. Defaults to 0x1000.
- `--stack reserve`
- `--stack reserve,commit`
Specify the amount of memory to reserve (and, optionally, commit) to be used as stack for this program. The default is 32MB reserved, 4K committed.
- `--subsystem which`
- `--subsystem which:major`
- `--subsystem which:major.minor`
Specifies the subsystem under which a program will execute, the legal values for *which* are `native`, `windows`, `console`, and `posix`. You may optionally also set the subsystem version.
- `--warn-duplicate-exports`
Emit warnings when duplicated export directives are encountered.

ld Environment Variables

You can change the behavior of `ld` with the environment variables, `GNUTARGET` and

`LDEMULATION`, and `COLLECT_NO_DEMANGLE`.

`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 “BFD Library” on page 67). 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.

`LDEMULATION` determines the default emulation if you don't use the `-m` option. The emulation can affect various aspects of linker behaviour, particularly the default linker script. You can list the available emulations with the `--verbose` or `-v` options. If the `-m` option is not used, and the `LDEMULATION` environment variable is not defined, the default emulation depends upon how the linker was configured.

Normally, the linker will default to demangling symbols. However, if `COLLECT_NO_DEMANGLE` is set in the environment, then it will default to not demangling symbols. This environment variable, `COLLECT_NO_DEMANGLE`, is used in a similar fashion by the `gcc` linker wrapper program. The default may be overridden by the `--demangle` and `--no-demangle` options.

3

Linker Scripts

A *linker script* controls every link. Such a script derives from the linker command language. The main purpose of the linker script is to describe how the sections in the input files should map into the output file, which controls the memory layout of the output file. However, when necessary, the linker script can also direct the linker to perform many other operations, using the linker commands.

The following documentation discusses the fundamentals of the linker script.

- “Basic Linker Script Concepts” on page 30
- “Linker Script Format” on page 31
- “Simple Linker Script Example” on page 31
- “Simple Linker Script Commands” on page 32
- “Assigning Values to Symbols” on page 35

- “SECTIONS Command” on page 37
- “MEMORY Command” on page 49
- “PHDRS Command” on page 50
- “VERSION Command” on page 53
- “Expressions in Linker Scripts” on page 55
- “Implicit Linker Scripts” on page 61

Basic Linker Script Concepts

The following documentation discusses some basic concepts and vocabulary in order to describe the linker script language.

The linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that compiles into the linker executable. You can use the `--verbose` command line option to display the default linker script. Certain command line options, such as `-r` or `-N`, will affect the default linker script.

You may supply your own linker script by using the `-T` command line option. When you do this, your linker script will replace the default linker script.

You may also use linker scripts implicitly by naming them as input files to the linker, as though they were files to be linked. See “Implicit Linker Scripts” on page 61.

The linker combines input files into a single output file. The output file and each input file are in a special data format known as an *object file format*. Each file is called an *object file*. The output file is often called an *executable*, but for our purposes it is also called an object file. Each object file has, among other things, a list of *sections*. A section in an input file is sometimes referred to as an *input section*; similarly, a section in the output file is an *output section*.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the *section contents*. A section may be marked as *loadable*, meaning that the contents should be loaded into memory when the output file is run. A section with no contents may be *allocatable*, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out). A section, which is neither loadable nor allocatable, typically contains some sort of debugging information.

Every loadable or allocatable output section has two addresses. The first is the *VMA*, or *virtual memory address*. This is the address the section will have when the output file is run. The second is the *LMA*, or *load memory address*. This is the address at which the section will be loaded. In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system). In this case the ROM address

would be the LMA, and the RAM address would be the VMA. You can see the sections in an object file by using the `objdump` program with the `-h` option.

Every object file also has a list of *symbols*, known as the *symbol table*. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable, which is referenced in the input file, will become an undefined symbol. You can see the symbols in an object file by using the `nm` program, or by using the `objdump` program with the `-t` option.

Linker Script Format

Linker scripts are text files. You write a linker script as a series of commands. Each command is either a keyword, possibly followed by arguments or an assignment to a symbol. You may separate commands using semicolons. Whitespace is generally ignored. Strings such as file or format names can normally be entered directly. If the file name contains a character such as a comma, which would otherwise serve to separate file names, you may put the file name in double quotes. There is no way to use a double quote character in a file name. You may include comments in linker scripts just as in C, delimited by `/*` and `*/`. As in C, comments are syntactically equivalent to whitespace.

Simple Linker Script Example

Many linker scripts are fairly simple. The simplest possible linker script has just one command: `SECTIONS`. You use the `SECTIONS` command to describe the memory layout of the output file. The `SECTIONS` command is a powerful command. Assume your program consists only of code, initialized data, and uninitialized data. These will be in the `.text`, `.data`, and `.bss` sections, respectively. Assume further that these are the only sections, which appear in your input files. For the following example, assume that the code should be loaded at address, `0x10000`, and that the data should start at address, `0x8000000`. The following linker script will do this function.

```
SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

You write the `SECTIONS` command as the keyword `SECTIONS`, followed by a series of symbol assignments and output section descriptions enclosed in curly braces. The first

line in the above example sets the special symbol, `.` (a period, which is the location counter). If you do not specify the address of an output section in some other way (other ways are described later), the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. The second line defines an output section, `.text`. The colon is required syntax, which may be ignored for now. Within the curly braces after the output section name, you list the names of the input sections, which should be placed into this output section. The `*` is a wildcard which matches any file name. The expression `*(.text)` means all `.text` input sections in all input files.

Since the location counter is `0x10000` when the output section `.text` is defined, the linker will set the address of the `.text` section in the output file to be `0x10000`. The remaining lines define the `.data` and `.bss` sections in the output file. The `.data` output section will be at address `0x8000000`. When the `.bss` output section is defined, the value of the location counter will be `0x8000000` plus the size of the `.data` output section. The effect is that the `.bss` output section will follow immediately after the `.data` output section in memory.

That is a complete linker script.

Simple Linker Script Commands

In the following documentation, the discussion describes the simple linker script commands. See also the complete descriptions of the command line options with “Using `ld` Command Line Options” on page 8 and, incidentally, the descriptions with “BFD Library” on page 67.

- “Setting the Entry Point” (on this page)
- “Commands Dealing with Files” on page 33
- “Commands Dealing with Object File Formats” on page 34
- “Other Linker Script Commands” on page 34

See also the complete descriptions for “Using `ld` Command Line Options” on page 8 and, incidentally, the descriptions for “BFD Library” on page 67.

Setting the Entry Point

The first instruction to execute in a program is called the *entry point*. You can use the `ENTRY` linker script command to set the entry point. The argument is a symbol name:

```
ENTRY (symbol)
```

There are several ways to set the entry point. The linker will set the entry point by trying each of the following methods in order, and stopping when one of them succeeds:

- The `-e entry` command-line option;

- The `ENTRY` (*symbol*) command in a linker script;
- The value of the symbol, `start`, if defined;
- The address of the first byte of the `.text` section, if present;
- The address, `0`.

Commands Dealing with Files

Several linker script commands deal with files. See also “Using `ld` Command Line Options” on page 8 and “BFD Library” on page 67.

`INCLUDE filename`

Include the linker script *filename* at this point. The file will be searched for in the current directory, and in any directory specified with the `-L` option. You can nest calls to `INCLUDE` up to 10 levels deep.

`INPUT (file, file, ...)`

`INPUT (file file ...)`

The `INPUT` command directs the linker to include the named files in the link, as though they were named on the command line. For example, if you always want to include `subr.o` any time you do a link, but you can not be bothered to put it on every link command line, then you can put `INPUT (subr.o)` in your linker script. In fact, if you like, you can list all of your input files in the linker script, and then invoke the linker with nothing but a `-T` option. The linker will first try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of `-L`. If you use `INPUT (-lfile)`, `ld` will transform the name to `libfile.a`, as with the command line argument `-l`. When you use the `INPUT` command in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

`GROUP(FILE, FILE, ...)`

`GROUP (file file ...)`

The `GROUP` 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.

`OUTPUT (filename)`

The `OUTPUT` command names the output file. Using `OUTPUT(FILENAME)` in the linker script is exactly like using `-o filename` on the command line. If both are used, the command line option takes precedence. You can use the `OUTPUT` command to define a default name for the output file other than the usual default of `a.out`.

`SEARCH_DIR (path)`

The `SEARCH_DIR` command adds *path* to the list of paths where `ld` looks for archive libraries. Using `SEARCH_DIR (path)` is exactly like using `-L path` on the command line; see “Using `ld` Command Line Options” on page 8. If both are

used, then the linker will search both paths. Paths specified using the command line option are searched first.

`STARTUP (filename)`

The `STARTUP` command is just like the `INPUT` command, except that *filename* will become the first input file to be linked, as though it were specified first on the command line. This may be useful when using a system in which the entry point is always the start of the first file.

Commands Dealing with Object File Formats

A couple of linker script commands deal with object file formats. See also “Using `ld` Command Line Options” on page 8 and “BFD Library” on page 67.

`OUTPUT_FORMAT (bfdname)`

`OUTPUT_FORMAT(default, big, little)`

The `OUTPUT_FORMAT` command names which BFD format to use for the output file. Using `OUTPUT_FORMAT (bfdname)` is exactly like using `-oformat bfdname` on the command line. If both are used, the command line option takes precedence.

You can use `OUTPUT_FORMAT` with three arguments to use different formats based on the `-EB` and `-EL` command line options. This permits the linker script to set the output format based on the desired endianness. If neither `-EB` nor `-EL` is used, then the output format will be the first argument, `DEFAULT`. If `-EB` is used, the output format will be the second argument, `BIG`. If `-EL` is used, the output format will be the third argument, `LITTLE`. For example, the default linker script for the MIPS ELF target uses the following command:

`OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)`

This says that the default format for the output file is `elf32-bigmips`, but if the user uses the `-EL` command line option, the output file will be created in the `elf32-littlemips` format.

`TARGET (bfdname)`The `TARGET` command names which BFD format to use when reading input files. It affects subsequent `INPUT` and `GROUP` commands. This command is like using `-b bfdname` on the command line. If the `TARGET` command is used but `OUTPUT_FORMAT` is not, then the last `TARGET` command is also used to set the format for the output file.

Other Linker Script Commands

There are a few other linker scripts commands. See also “Using `ld` Command Line Options” on page 8 and “BFD Library” on page 67.

`ASSERT (exp, message)`

Ensure that *exp* is non-zero. If it is zero, then exit the linker with an error code, and print *message*.

`EXTERN (symbol 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. You may list several *symbols* for each `EXTERN`, and you may use `EXTERN` multiple times. This command has the same effect as the `-u` command-line option.

`FORCE_COMMON_ALLOCATION`

Same effect as the `-d` command-line option, making `ld` assign space to common symbols even if a relocatable output file is specified (`-r`).

`NOCROSSREFS(section section ...)`

Tells `ld` to issue an error about any references among certain output sections.

In certain types of programs, particularly on embedded systems when using overlays, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors. For example, it would be an error if code in one section called a function defined in the other section. The `NOCROSSREFS` command takes a list of output section names. If `ld` detects any cross-references between the sections, it reports an error and returns a non-zero exit status. Remember that the `NOCROSSREFS` command uses output section names, not input section names.

`OUTPUT_ARCH(bfdarch)`

Specify a particular output machine architecture, *bfdarch*. The argument is one of the names used by the BFD library. You can see the architecture of an object file by using the `objdump` program with the `-f` option.

Assigning Values to Symbols

You may assign a value to a symbol in a linker script. This will define the symbol as a global symbol. The following documentation discusses such assignments in more detail.

- “Simple Assignments” on page 35
- “PROVIDE Keyword” on page 36

Simple Assignments

You may assign to a symbol using any of the C assignment operators:

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

- The first case will define *symbol* to the value of *expression*. In the other cases, *symbol* must already be defined, and the value will be accordingly adjusted.

- The special `.` symbol name indicates the location counter. You may only use this within a `SECTIONS` command.
- The semicolon after *expression* is required.
- See “Expressions in Linker Scripts” on page 55.
- You may write symbol assignments as commands in their own right, or as statements within a `SECTIONS` command, or as part of an output section description in a `SECTIONS` command.
- The section of the symbol will be set from the section of the expression; for more information, see “Expressions in Linker Scripts” on page 55.
- The following is an example showing the three different places that symbol assignments may be used:

```
floating_point = 0;
SECTIONS
{
    .text :
    {
        *(.text)
        _etext = .;
    }
    _bdata = (. + 3) & ~ 4;
    .data : { *(.data) }
}
```

In the previous example, the `floating_point` symbol will be defined as zero. The `_etext` symbol will be defined as the address following the last `.text` input section. The symbol `_bdata` will be defined as the address following the `.text` output section aligned upward to a 4 byte boundary.

PROVIDE Keyword

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and 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)`.

Here is an example of using `PROVIDE` to define `etext`:

```
SECTIONS
{
    .text :
    {
        *(.text)
        _etext = .;
        PROVIDE(etext = .);
    }
}
```

```
}
```

In the previous example, if the program defines `_etext`, the linker will give a multiple definition error. If, on the other hand, the program defines `etext`, the linker will silently use the definition in the program. If the program references `etext` but does not define it, the linker will use the definition in the linker script.

SECTIONS Command

The `SECTIONS` command tells the linker how to map input sections into output sections, and how to place the output sections in memory. The following documentation describes more of the `SECTIONS` command.

- “Output Section Description” on page 38
- “Output Section Name” on page 38
- “Output Section Address” on page 39
- “Input Section Description” on page 39
- “Input Section Basics” on page 39
- “Input Section Wildcard Patterns” on page 40
- “Input Section for Common Symbols” on page 41
- “Input Section and Garbage Collection” on page 42
- “Input Section Example” on page 42
- “Output Section Data” on page 42
- “Output Section Keywords” on page 43
- “Output Section Discarding” on page 44
- “Output Section Attributes” on page 45
- “Output Section Type” on page 45
- “Output Section LMA” on page 45
- “Output Section Region” on page 46
- “Output Section to Programs Previously Defined” on page 46
- “Output Section Fill” on page 47
- “Overlay Description” on page 47

The format of the `SECTIONS` command is:

```
SECTIONS
{
    sections-command
    sections-command
    ...
}
```

Each *sections-command* may be one of the following:

- An `ENTRY` command (see “Setting the Entry Point” on page 32)
- A symbol assignment (see “Simple Assignments” on page 35)
- An output section description (see “Output Section Description” on page 38)
- An overlay description (see “Overlay Description” on page 47)

The `ENTRY` command and symbol assignments are permitted inside the `SECTIONS` command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because you can use those commands at meaningful points in the layout of the output file. See “Output Section Description” on page 38 and “Overlay Description” on page 47.

If you do not use a `SECTIONS` command in your linker script, the linker will place 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. The first section will be at address-zero.

Output Section Description

The full description of an output section looks like this:

```
SECTION [address] [(type)] : [AT(LMA)]
{
    output-sections-command
    output-sections-command
    ...
} [>region] [:phdr :phdr ...] [=fillexp]
```

Most output sections do not use most of the optional section attributes. The whitespace around `SECTION` is required, so that the section name is unambiguous. The colon and the curly braces are also required. The line breaks and other white space are optional.

Each *output-sections-command* may be one of the following:

- A symbol assignment (see “Simple Assignments” on page 35)
- An input section description (see “Input Section Description” on page 39)
- Data values to include directly (see “Output Section Data” on page 42)
- A special output section keyword (see “Output Section Keywords” on page 43)

Output Section Name

The name of the output section is *section*. *section* 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 a name, which contains any unusual characters such as commas, must be quoted. The output section name `/DISCARD/` is special. See “Output Section Discarding” on page 44.

Output Section Address

The *address* is an expression for the VMA (the virtual memory address) of the output section. If you do not provide *address*, the linker will set it based on `REGION` if present, or otherwise based on the current value of the location counter.

If you provide *address*, the address of the output section will be set to precisely that specification. If you provide neither *address* nor *region*, then the address of the output section will be set to the current value of the location counter aligned to the alignment requirements of the output section.

The alignment requirement of the output section is the strictest alignment of any input section contained within the output section. For example `.text . : { *(.text) }` and `.text : { *(.text) }` are subtly different. The first will set the address of the `.text` output section to the current value of the location counter. The second will set it to the current value of the location counter aligned to the strictest alignment of a `.text` input section. The address may be an arbitrary expression. See “Expressions in Linker Scripts” on page 55. For example, if you want to align the section on a 0x10 byte boundary, so that the lowest four bits of the section address are zero, you could use something like the following declaration:

```
.text ALIGN(0x10) : { *(.text) }
```

This declaration works because `ALIGN` returns the current location counter aligned upward to the specified value. Specifying an address for a section will change the value of the location counter.

Input Section Description

The most common output section command is an *input section description*. The input section description is the most basic linker script operation. You use output sections to tell the linker how to lay out your program in memory. You use input section descriptions to tell the linker how to map the input files into your memory layout.

Input Section Basics

An *input section description* consists of a file name optionally followed by a list of section names in parentheses. The file name and the section name may be wildcard patterns; see “Input Section Wildcard Patterns” on page 40. The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input `.text` sections, you would write:

```
*(.text)
```

The `*` is a wildcard which matches *any* file name.

There are two ways to include more than one section:

```
*(.text .rdata)
*(.text) *(.rdata)
```

The difference between these is the order in which the `.text` and `.rdata` input sections will appear in the output section. In the first example, they will be intermingled. In the second example, all `.text` input sections will appear first, followed by all `.rdata` input sections.

You can specify a file name to include sections from a particular file. You would do this if one or more of your files contain special data that needs to be at a particular location in memory. For example:

```
data.o(.data)
```

If you use a file name without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may be useful on occasion. For example:

```
data.o
```

When you use a file name, which does not contain any wild card characters, the linker will first see if you also specified the file name on the linker command line or in an `INPUT` command. If you did not, the linker will attempt to open the file as an input file, as though it appeared on the command line. Note that this differs from an `INPUT` command, because the linker will not search for the file in the archive search path.

Input Section Wildcard Patterns

In an input section description, either the file name or the section name or both may be wildcard patterns. The file name of `*` seen in many examples is a simple wildcard pattern for the file name. The wildcard patterns are like those used by the Unix shell.

`*`

Matches any number of characters.

`?`

Matches any single character.

`[chars]`

Matches a single instance of any of the *chars*; the `-` character may be used to specify a range of characters, as in `[a-z]` to match any lower case letter.

`\`

Quotes the following character.

When a file name is matched with a wildcard, the wildcard characters will not match a `/` character (used to separate directory names on Unix). A pattern consisting of a single `*` character is an exception; it will always match any file name, whether it contains a `/` or not. In a section name, the wildcard characters will match a `/` character.

File name wildcard patterns only match files which are explicitly specified on the

command line or in an `INPUT` command. The linker does not search directories to expand wildcards.

If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker will use the first match in the linker script. For example, this sequence of input section descriptions is probably in error, because the `data.o` rule will not be used:

```
.data : { *(.data) }
.data1 : { data.o(.data) }
```

Normally, the linker will place files and sections matched by wildcards in the order in which they are seen during the link. You can change this by using the `SORT` keyword, which appears before a wildcard pattern in parentheses (such as `SORT(.text*)`).

When the `SORT` keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

If you ever get confused about where input sections are going, use the `-M` linker option to generate a map file. The map file shows precisely, how input sections are mapped to output sections.

The following example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all `.text` sections in `.text` and all `.bss` sections in `.bss`. The linker will place the `.data` section from all files beginning with an upper case character in `.DATA`; for all other files, the linker will place the `.data` section in `.data`.

```
SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Input Section for Common Symbols

A special notation is needed for common symbols, because in many object-file formats common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named `COMMON`.

You may use file names with the `COMMON` section just as with any other input sections. You can use this to place common symbols from a particular input file in one section while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the `.bss` section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

Some object file formats have more than one type of common symbol. For example, the MIPS ELF object file format distinguishes standard common symbols and small common symbols. In this case, the linker will use a different special section name for

other types of common symbols. In the case of MIPS ELF, the linker uses `COMMON` for standard common symbols and `.scommon` for small common symbols. This permits you to map the different types of common symbols into memory at different locations.

You will sometimes see `[COMMON]` in old linker scripts. This notation is now considered obsolete. It is equivalent to `*(COMMON)`.

Input Section and Garbage Collection

When link-time garbage collection is in use (`--gc-sections`), it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with `KEEP()`, as in `KEEP(*(.init))` or `KEEP(SORT(*)(.ctors))`.

Input Section Example

The following example is a complete linker script. It tells the linker to read all of the sections from file `all.o` and place 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)
    }
}
```

Output Section Data

You can include explicit bytes of data in an output section by using `BYTE`, `SHORT`, `LONG`, `QUAD`, or `SQUAD` as an output section command. Each keyword is followed by an expression in parentheses providing the value to store; see “Expressions in Linker Scripts” on page 55. The value of the expression is stored at the current value of the location counter.

The `BYTE`, `SHORT`, `LONG`, and `QUAD` commands store one, two, four, and eight bytes (respectively). After storing the bytes, the location counter is incremented by the number of bytes stored. For example, this will store the byte 1 followed by the four byte value of the symbol, `addr`:

```
BYTE(1)
LONG(addr)
```

When using a 64-bit host or target, `QUAD` and `SQUAD` are the same; they both store an 8-byte, or 64-bit, value. When both host and target are 32 bits, an expression is computed as 32 bits. In this case `QUAD` stores a 32-bit value zero extended to 64 bits, and `SQUAD` stores a 32-bit value sign extended to 64 bits.

If the object file format of the output file has an explicit endianness, which is the normal case, the value will be stored in that endianness. When the object file format does not have an explicit endianness, as is true of, for example, S-records, the value will be stored in the endianness of the first input object file.

You may use the `FILL` command to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the two least significant bytes of the expression, repeated as necessary. A `FILL` statement covers memory locations after the point at which 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.

The following example shows how to fill unspecified regions of memory with the value `0x9090`:

```
FILL(0x9090)
```

The `FILL` command is similar to the `=fillexp` output section attribute (see “Output Section Fill” on page 47); but it only affects the part of the section following the `FILL` command, rather than the entire section. If both are used, the `FILL` command takes precedence.

Output Section Keywords

There are a couple of keywords, which can appear as output section commands.

`CREATE_OBJECT_SYMBOLS`

The command tells the linker to create a symbol for each input file. The name of each symbol will be the name of the corresponding input file. The section of each symbol will be the output section in which the `CREATE_OBJECT_SYMBOLS` command appears.

This is conventional for the `a.out` object file format. It is not normally used for any other object file format.

`CONSTRUCTORS`

When linking, using the `a.out` object file format, the linker uses an unusual set construct to support C++ global constructors and destructors. When linking object

file formats, which do not support arbitrary sections, such as `ECOFF` and `XCOFF`, the linker will automatically recognize C++ global constructors and destructors by name. For these object file formats, the `CONSTRUCTORS` command tells the linker to place constructor information in the output section where the `CONSTRUCTORS` command appears. The `CONSTRUCTORS` command is ignored for other object file formats. The symbol `__CTOR_LIST__` marks the start of the global constructors, and the symbol `__DTOR_LIST` marks the end. The first word in the list is the number of entries, followed by the address of each constructor or destructor, followed by a zero word. The compiler must arrange to actually run the code. For these object file formats GNU C++ normally calls constructors from a subroutine, `__main`; a call to `__main` is automatically inserted into the startup code for `main`. GNU C++ normally runs destructors either by using `atexit`, or directly from the function `exit`. For object file formats such as `COFF` or `ELF`, which support arbitrary section names, GNU C++ will normally arrange to put the addresses of global constructors and destructors into the `.ctors` and `.dtors` sections. Placing the following sequence into your linker script will build the sort of table that the GNU C++ runtime code expects to see.

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
*(.dtors)
LONG(0)
__DTOR_END__ = .;
```

Normally the compiler and linker will handle these issues automatically, and you will not need to concern yourself with them. However, you may need to consider this occurrence, if you are using C++ and writing your own linker scripts.

Output Section Discarding

The linker will not create output section which do not have any contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. For example, the `.foo { *(.foo) }` declaration will only create a `.foo` section in the output file if there is a `.foo` section in at least one input file. If you use anything other than an input section description as an output section command, such as a symbol assignment, then the output section will always be created, even if there are no matching input sections. The special output section name, `/DISCARD/`, may be used to discard input sections. Any input sections assigned to an output section named `/DISCARD/` are not included in the output file.

Output Section Attributes

A full description of an output section looked like the following example (see also “Output Section Description” on page 38).

```
SECTION [address] [(type)] : [AT(LMA)]
{
    output-sections-command
    output-sections-command
    ...
} [>region] [:phdr :phdr ...] [=fillexp]
```

See the following documentation for descriptions of the remaining output section attributes.

- “Output Section Type” on page 45
- “Output Section LMA” on page 45
- “Output Section Region” on page 46
- “Output Section to Programs Previously Defined” on page 46
- “Output Section Fill” on page 47

Output Section Type

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

NOLOAD

The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.

DSECT

COPY

INFO

OVERLAY

These type names are supported for backward compatibility, and are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.

The linker normally sets the attributes of an output section, based on the input sections, which map into it. You can override this by using the section type. For example, in the script sample below, the ROM section is addressed at memory location 0 and does not need to be loaded when the program is run. The contents of the ROM section will appear in the linker output file as usual.

```
SECTIONS {
    ROM 0 (NOLOAD) : { ... }
    ...
}
```

Output Section LMA

Every section has a virtual address (VMA) and a load address (LMA); see “Basic

Linker Script Concepts” on page 30. The address expression that, may appear in an output section description sets the VMA. The linker will normally set the LMA equal to the VMA. You can change that by using the `AT` keyword. The expression, LMA, that follows the `AT` keyword specifies the load address of the section. This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called `.text`, which starts at `0x1000`, one called `.mdata`, which is loaded at the end of the `.text` section even though its VMA is `0x2000`, and one called `.bss` to hold uninitialized data at address, `0x3000`. The symbol `_data` is defined with the value, `0x2000`, which shows that the location counter holds the VMA value, not the LMA value.

```
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 use with a program generated with this linker script would include something like the following example shows, copying the initialized data from the ROM image to its runtime address. This code takes advantage of the symbols defined by the linker script.

```
extern char _etext, _data, _edata, _bstart, _bend;
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;
```

Output Section Region

You can assign a section to a previously defined region of memory by using `>REGION`. The following example shows the way.

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

Output Section to Programs Previously Defined

You can assign a section to a previously defined program segment by using `:phdr`. If a section is assigned to one or more segments, then all subsequent allocated sections

will be assigned to those segments as well, unless they use an explicitly `:phdr` modifier. To prevent a section from being assigned to a segment when it would normally default to one, use `:NONE`. See “PHDRS Command” on page 50. The following example shows the way.

```
PHDRS { text PT_LOAD ; }
SECTIONS { .text : { *(.text) } :text }
```

Output Section Fill

You can set the fill pattern for an entire section by using `=fillexp`. `fillexp` is an expression; see “Expressions in Linker Scripts” on page 55. Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) 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` command in the output section commands. See “Output Section Data” on page 42. The following example shows the way.

```
SECTIONS { .text : { *(.text) } =0x9090 }
```

Overlay Description

An overlay description provides an easy way to describe sections, which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the runtime memory address as required, perhaps by simply manipulating addressing bits. This approach can be useful, for example, when a certain region of memory is faster than another region of memory.

Overlays are described using the `OVERLAY` command. The `OVERLAY` command is used within a `SECTIONS` command, like an output section description. The full syntax of the `OVERLAY` command is shown in the following example.

```
OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
{
    secname1
    {
        output-section-command
output-section-command
        ...
    } [[:PHDR...]] [=FILL]
    secname2
    {
        output-section-command
output-section-command
        ...
    } [[:phdr...]] [=fill]
    ...
} [>region] [[:phdr...]] [=fill]
```

Everything is optional except `OVERLAY` (a keyword), and each section must have a name (as in the previous example, `secname1` and `secname2`). The section definitions within the `OVERLAY` construct are identical to those within the general `SECTIONS` construct, except that no addresses and no memory regions may be defined for sections within an `OVERLAY`. See “`SECTIONS` Command” on page 37.

The sections are all defined with the same starting address. The load addresses of the sections are arranged, so that they are consecutive in memory, starting at the load address used for the `OVERLAY` as a whole (as with normal section definitions. The load address is optional, and defaults to the start address. The start address is also optional, and defaults to the current value of the location counter).

If the `NOCROSSREFS` keyword is used, and there are any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another.

For each section within the `OVERLAY`, the linker automatically defines two symbols. The symbol `__load_start_secname` is defined as the starting load address of the section. The symbol `__load_stop_secname` is defined as the final load address of the section. Any characters within `secname` that are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary. At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section. The following example shows the way. Remember that this would appear inside a `SECTIONS` construct.

```
OVERLAY 0x1000 : AT (0x4000)
{
    .text0 { o1/*.o(.text) }
    .text1 { o2/*.o(.text) }
}
```

This will define both `.text0` and `.text1` to start at address, `0x1000`. `.text0` will be loaded at address, `0x4000`, and `.text1` will be loaded immediately after `.text0`. The following symbols will be defined: `__load_start_text0`, `__load_stop_text0`, `__load_start_text1`, `__load_stop_text1`. C code to copy overlay `.text1` into the overlay area might look like the following example’s code.

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

Everything that the `OVERLAY` command does can be done using the more basic commands. The previous example could have been written identically as follows.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
```

```
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

MEMORY Command

The linker’s default configuration permits allocation of all available memory. You can override this by using the `MEMORY` command.

The `MEMORY` command describes the location and size of blocks of memory in the target. You can use it to describe which memory regions may be used by the linker, and which memory regions it must avoid. You can then assign sections to particular memory regions. The linker will set section addresses based on the memory regions, and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

A linker script 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 like the following example shows.

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
}
```

name is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name.

The *attr* string is an optional list of attributes that specify whether to use a particular memory region for an input section, which is not explicitly mapped in the linker script. If you do not specify an output section for some input section, the linker will create an output section with the same name as the input section. If you define region attributes, the linker will use them to select the memory region for the output section that it creates. See “SECTIONS Command” on page 37. The *attr* string must consist only of the following characters.

```
R      Read-only section
W      Read/write section
X      Executable section
A      Allocatable section
I      Initialized section
```

L
Same as I

!
Invert the sense of any of the preceding attributes

If an unmapped section matches any of the listed attributes other than `!`, it will be placed in the memory region. The `!` attribute reverses this test, so that an unmapped section will be placed in the memory region only if it does not match any of the listed attributes.

The `ORIGIN` is an expression for the start address of the memory region. The expression must evaluate to a constant before memory allocation is performed, which means that you may not use any section relative symbols. The `ORIGIN` keyword may be abbreviated to `org` or `o` (but not, for example, `ORG`).

The `len` is an expression for the size in bytes of the memory region. As with the `origin` expression, the expression must evaluate to a constant before memory allocation is performed. The `LENGTH` keyword may be abbreviated to `len` or `l`.

In the following example, there are two memory regions available for allocation: one starting at 0 for 256 kilobytes, and the other starting at 0x40000000 for four megabytes. The linker will place into the `rom` memory region every section, which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections, which are not explicitly mapped into a memory region into the `ram` memory region.

```
MEMORY
{
  rom (rx) : ORIGIN = 0, LENGTH = 256K
  ram (!rx) : org = 0x40000000, l = 4M
}
```

Once you define a memory region, you can direct the linker to place specific output sections into that memory region by using the `>region` output section attribute. For example, if you have a memory region named `mem`, you would use `>mem` in the output section definition. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message. See “Output Section Region” on page 46.

PHDRS Command

The ELF object file format uses *program headers*, also known as *segments*. The program headers describe how the program should be loaded into memory. You can print them out by using the `objdump` program with the `-p` option. When you run an ELF program on a native ELF system, the system loader reads the program headers in order to figure out how to load the program. This will only work if the program

headers are set correctly. This documentation does not describe the details of how the system loader interprets program headers; for more information, see the ELF ABI.

The linker will create reasonable program headers by default. However, in some cases, you may need to specify the program headers more precisely. You may use the `PHDRS` command for this purpose. When the linker sees the `PHDRS` command in the linker script, it will not create any program headers other than the ones specified.

The linker only pays attention to the `PHDRS` command when generating an ELF output file. In other cases, the linker will simply ignore `PHDRS`.

The following example shows the syntax of the `PHDRS` command. The words, `PHDRS`, `FILEHDR`, `AT`, and `FLAGS`, are keywords.

```
PHDRS
{
    name type[ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
        [ FLAGS ( flags ) ] ;
}
```

The *name* is used only for reference in the `SECTIONS` command of the linker script. It is not put into the output file. Program header names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each program header must have a distinct name.

Certain program header types describe segments of memory, which the system loader will load from the file. In the linker script, you specify the contents of these segments by placing allocatable output sections in the segments. You use the `:phdr` output section attribute to place a section in a particular segment. See “Output Section to Programs Previously Defined” on page 46.

It is normal to put certain sections in more than one segment. This merely implies that one segment of memory contains another. You may repeat `:phdr`, using it once for each segment which should contain the section.

If you place a section in one or more segments using `:phdr`, then the linker will place all subsequent allocatable sections which do not specify `:phdr` in the same segments. This is for convenience, since generally a whole set of contiguous sections will be placed in a single segment. To prevent a section from being assigned to a segment when it would normally default to one, use `:NONE`.

You may use the `FILEHDR` and `PHDRS` keywords appear after the program header type to further describe the contents of the segment. The `FILEHDR` keyword means that the segment should include the ELF file header. The `PHDRS` keyword means that the segment should include the ELF program headers themselves.

type may be one of the following, the numbers indicating the value of the keyword.

- `PT_NULL (0)` indicates an unused program header.
- `PT_LOAD (1)` indicates that this program header describes a segment to be loaded from the file.

- `PT_DYNAMIC` (2) indicates a segment where dynamic linking information can be found.
- `PT_INTERP` (3) indicates a segment where the name of the program interpreter may be found.
- `PT_NOTE` (4) indicates a segment holding note information.
- `PT_SHLIB` (5) is a reserved program header type, defined but not specified by the ELF ABI.
- `PT_PHDR` (6) indicates a segment where the program headers may be found.
- *expression* is an expression giving the numeric type of the program header. This may be used for types not defined above.

You can specify that a segment should be loaded at a particular address in memory by using an `AT` expression. This is identical to the `AT` command used as an output section attribute. The `AT` command for a program header, overrides the output section attribute. See “Output Section LMA” on page 45.

The linker will normally set the segment flags based on the sections, which comprise the segment. You may use the `FLAGS` keyword to explicitly specify the segment flags. The value of *flags* must be an integer. It is used to set the `p_flags` field of the program header. The following example shows the use of `PHDRS` with a typical set of program headers used on a native ELF system.

```
PHDRS
{
    headers PT_PHDR PHDRS ;
    interp PT_INTERP ;
    text PT_LOAD FILEHDR PHDRS ;
    data PT_LOAD ;
    dynamic PT_DYNAMIC ;
}

SECTIONS
{
    . = SIZEOF_HEADERS;
    .interp : { *(.interp) } :text :interp
    .text : { *(.text) } :text
    .rodata : { *(.rodata) } /* defaults to :text */
    ...
    . = . + 0x1000; /* move to a new page in memory */
    .data : { *(.data) } :data
    .dynamic : { *(.dynamic) } :data :dynamic
    ...
}
```

VERSION Command

The linker supports symbol versions when using ELF. Symbol versions are only useful when using shared libraries. The dynamic linker can use symbol versions to select a specific version of a function when it runs a program that may have been linked against an earlier version of the shared library.

You can include a version script directly in the main linker script, or you can supply the version script as an implicit linker script. You can also use the `--version-script` linker option.

The syntax of the `VERSION` command follows.

```
VERSION { version-script-commands }
```

The format of the version script commands is identical to that used by Sun's linker in Solaris 2.5. The version script defines a tree of version nodes. You specify the node names and interdependencies in the version script. You can specify which symbols are bound to which version nodes, and you can reduce a specified set of symbols to local scope so that they are not globally visible outside of the shared library.

The easiest way to demonstrate the version script language is with the following example.

```
VERS_1.1 {
    global:
    fool;
    local:
    old*;
    original*;
    new*;
};

VERS_1.2 {
    foo2;
} VERS_1.1;

VERS_2.0 {
    bar1; bar2;
} VERS_1.2;
```

This example version script defines three version nodes. The first version node defined is `VERS_1.1`; it has no other dependencies. The script binds the symbol, `foo1`, to `VERS_1.1`. It reduces a number of symbols to local scope so that they are not visible outside of the shared library.

Next, the version script defines node, `VERS_1.2`. This node depends upon `VERS_1.1`. The script binds the symbol, `foo2`, to the version node, `VERS_1.2`.

Finally, the version script defines node `VERS_2.0`. This node depends upon `VERS_1.2`. The script binds the symbols, `bar1` and `bar2`, to the version node, `VERS_2.0`.

When the linker finds a symbol defined in a library, which is not specifically bound to a version node, it will effectively bind it to an unspecified base version of the library. You can bind all otherwise unspecified symbols to a given version node by using `global: *` somewhere in the version script.

The names of the version nodes have no specific meaning other than what they might suggest to the person reading them. The 2.0 version could just as well have appeared in between 1.1 and 1.2. However, this would be a confusing way to write a version script.

When you link an application against a shared library that has versioned symbols, the application itself knows which version of each symbol it requires, and it also knows which version nodes it needs from each shared library it is linked against. Thus at runtime, the dynamic loader can make a quick check to make sure that the libraries you have linked against do in fact supply all of the version nodes that the application will need to resolve all of the dynamic symbols. In this way it is possible for the dynamic linker to know with certainty that all external symbols that it needs will be resolvable without having to search for each symbol reference.

The symbol versioning is in effect a much more sophisticated way of doing minor version checking that SunOS does. The fundamental problem that is being addressed here is that typically references to external functions are bound on an as-needed basis, and are not all bound when the application starts up. If a shared library is out of date, a required interface may be missing; when the application tries to use that interface, it may suddenly and unexpectedly fail. With symbol versioning, the user will get a warning when they start their program if the libraries being used with the application are too old.

There are several GNU extensions to Sun's versioning approach. The first of these is the ability to bind a symbol to a version node in the source file where the symbol is defined instead of in the versioning script. This was done mainly to reduce the burden on the library maintainer. You can do this by putting something like this in the C source file:

```
__asm__( ".symver original_foo,foo@VERS_1.1" );
```

This renames the function, `original_foo`, to be an alias for `foo`, bound to the version node, `VERS_1.1`. The `local:` directive can be used to prevent the symbol `original_foo` from being exported.

The second GNU extension is to allow multiple versions of the same function to appear in a given, shared library. In this way you can make an incompatible change to an interface without increasing the major version number of the shared library, while still allowing applications linked against the old interface to continue to function.

To do this, you must use multiple `.symver` directives in the source file. Here is an example:

```
__asm__( ".symver original_foo,foo@" );  
__asm__( ".symver old_foo,foo@VERS_1.1" );
```

```
__asm__( ".symver old_fool,foo@VERS_1.2" );
__asm__( ".symver new_foo,foo@@VERS_2.0" );
```

In this example, `foo@` represents the symbol, `foo`, bound to the unspecified base version of the symbol. The source file that contains this example would define four C functions: `original_foo`, `old_foo`, `old_fool`, and `new_foo`.

When you have multiple definitions of a given symbol, there needs to be some way to specify a default version to which external references to this symbol will be bound. You can do this with the `foo@@VERS_2.0` type of `.symver` directive. You can only declare one version of a symbol as the default in this manner; otherwise you would effectively have multiple definitions of the same symbol.

If you wish to bind a reference to a specific version of the symbol within the shared library, you can use the aliases of convenience (for instance, `old_foo`), or you can use the `.symver` directive to specifically bind to an external version of the function in question.

Expressions in Linker Scripts

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as integers. All expressions are evaluated in the same size, which is 32 bits if both the host and target are 32 bits, and is otherwise 64 bits. You can use and set symbol values in expressions. The linker defines several special purpose builtin functions for use in expressions. See the following documentation for more details.

- “Constants” on page 55
- “Symbol Names” on page 56
- “The Location Counter” on page 56
- “Operators” on page 57
- “Evaluation” on page 57
- “The Section of an Expression” on page 58
- “Builtin Functions” on page 58

Constants

All constants are integers. As in C, the linker considers an integer beginning with 0 to be octal, and an integer beginning with `0x` or `0X` to be hexadecimal.

The linker considers other integers to be decimal.

In addition, you can use the suffixes, `K` and `M`, to scale a constant by `1024` or `1024*1024`, respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods, 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.

The Location Counter

The special linker *dot* (`.`) variable always contains the current output location counter. Since the `.` always refers to a location in an output section, it may only appear in an expression within a `SECTIONS` command. The `.` symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to `.` 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, the `.text` section from `file1` is located at the beginning of the output section `output`. It is followed by a 1000 byte gap. Then the `.text` section from `file2` appears, also with a 1000 byte gap following before the `.text` section from `file3`. The notation `= 0x1234` specifies data to write in the gaps.

Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels; see Table 1.

Table 1: Arithmetic operators with precedence levels and bindings associations

<i>Precedence</i>	<i>Association</i>	<i>Operators</i>	<i>Notes</i>
(highest)			
1	left	! - ~	†
2	left	* / %	
3	left	+ -	
4	left	>> <<	
5	left	== != > < <= >=	
6	left	&	
7	left		
8	left	&&	
9	left		
10	right	? :	
11	right	&= += -= *= /=	‡
(lowest)			

† Prefix operators

‡ See “Assigning Values to Symbols” on page 35.

Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and 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 linker script. 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.

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 must be evaluated during section allocation.

If the result of an expression is required, but the value is not available, then an error results. The following example shows a script that can cause the error message.

```
SECTIONS
{
    .text 9+this_isnt_constant :
```

```
    { *(.text) }  
}
```

A “non constant expression for initial address” message would result.

The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.

The position of the expression within the linker script determines whether it is absolute or relative. An expression, which appears within an output section definition, is relative to the base of the output section. An expression, which appears elsewhere, will be absolute.

A symbol set to a relative expression will be relocatable if you request relocatable output using the `-r` option. That means that a further link operation may change the value of the symbol. The symbol’s section will be the section of the relative expression.

A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.

You can use the builtin function `ABSOLUTE` to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section `.data`:

```
SECTIONS  
{  
    .data : { *(.data) _edata = ABSOLUTE(.); }  
}
```

If `ABSOLUTE` were not used, `_edata` would be relative to the `.data` section.

Builtin Functions

The linker script language includes the following builtin functions for use in linker script expressions.

`ABSOLUTE(exp)`

Return the absolute (non-relocatable, as opposed to non-negative) value of the *exp* expression. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative. See “Expressions in Linker Scripts” on page 55.

`ADDR(section)`

Return the absolute address (the VMA) 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 location counter (.) aligned to the next *exp* boundary. *exp* must be an expression whose value is a power of two. This is equivalent to:

$$(. + exp - 1) \& \sim(exp - 1)$$

ALIGN does not change the value of the location counter, it just does arithmetic on it. Here is an example which aligns the output `.data` section to the next `0x2000` byte boundary after the preceding section and sets 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 ADDRESS attribute of a section definition. The second use of ALIGN is to define the value of a symbol. The builtin function NEXT is closely related to ALIGN. See “Output Section Address” on page 39.

BLOCK(*exp*)

This is a synonym for ALIGN, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

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 script 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 : . ;
    ...
    }
    ...
}

```

`LOADADDR(section)`

Return the absolute LMA of the named `SECTION`. This is normally the same as `ADDR`, but it may be different if the `AT` attribute is used in the output section definition.

`MAX(exp1, exp2)`

Returns the maximum of *exp1* and *exp2*.

`MIN(exp1, exp2)`

Returns the minimum of *exp1* and *exp2*.

`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. If the section has not been allocated when this is evaluated, the linker will report an error. See “PHDRS Command” on page 50. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```

SECTIONS{ ...
    .output {
        .start = . ;
    ...
}

```

`SIZEOF_HEADERS`

Return the size in bytes of the output file’s headers. This is information which appears at the start of the output file. You can use this number when setting the start address of the first section, if you choose, to facilitate paging.

When producing an ELF output file, if the linker script uses the `SIZEOF_HEADERS` builtin function, the linker must compute the number of program headers before it has determined all the section addresses and sizes. If the linker later discovers that it needs additional program headers, it will report an “not enough room for program headers” error. To avoid this error, you must avoid using the `SIZEOF_HEADERS` function, or you must rework your linker script to avoid forcing the linker to use additional program headers, or you must define the program headers yourself using the `PHDRS` command (see “PHDRS Command” on page 50).

Implicit Linker Scripts

If you specify a linker input file which the linker can not recognize as an object file or an archive file, it will try to read the file as a linker script. If the file can not be parsed as a linker script, the linker will report an error.

An implicit linker script will not replace the default linker script.

Typically an implicit linker script would contain only symbol assignments, or the `INPUT`, `GROUP`, or `VERSION` commands.

Any input files read because of an implicit linker script will be read at the position in the command line where the implicit linker script was read. This can affect archive searching.

4

ld Machine Dependent Features

The following documentation describes some machine independent features for the GNU linker.

- “ld and the H8/300 Processors” (below)
- “ld and Intel 960 Processors” on page 64
- “ld Support for Interworking Between ARM and Thumb Code” on page 65

Machines with ld having no additional functionality have no documentation.

ld and the H8/300 Processors

For the H8/300 processors, 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).

ld and Intel 960 Processors

You can use the `'-Architecture'` command line option to specify one of the two-letter names identifying members of the 960 processors; 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).

ld Support for Interworking Between ARM and Thumb Code

For the ARM, ld will generate code stubs to allow functions calls between ARM and Thumb code. These stubs only work with code that has been compiled and assembled with the `-mthumb-interwork` command line option. If it is necessary to link with old ARM object files or libraries, those which have not been compiled with the `-mthumb-interwork` option, then the `--support-old-code` command line switch should be given to the linker. This will make it generate larger stub functions which will work with non-interworking aware ARM code.

However, the linker does not support generating stubs for function calls to non-interworking aware Thumb code.

5

BFD Library

The linker accesses object and archive files using the BFD library (a library whose name comes from binary file descriptors).

The following documentation discusses the BFD library and how to use them.

- “How BFD Works (an Outline of BFD)” on page 68
- “Information Loss” on page 68
- “The BFD Canonical Object File Format” on page 69

The BFD library allows 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. To list all the formats available for your configuration, use `objdump -i` (see “objdump” in *Using binutils* in **GNUPro**

Auxiliary Development Tools).

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 “Information Loss” on page 68.

How BFD 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 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.

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 (such as `a.out`) or has sections without

names (such as 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.

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, by that stored by the canonical format, and by 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-files 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 big-endian and little-endian object files may be used together.
- *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).

6

MRI Compatible Script Files for the GNU Linker

The GNU linker is able to support the object files and linker scripts used by the the Microtech card C and C++ compilers. Microtech, or Microtech Research Incorporated are owned by Mneotr Graohics. MRI compatible linker scripts have a much simpler command set than the scripting language otherwise used with `ld`. `ld` supports the most commonly used MRI linker commands; these commands are described in the following documentation.

In general, MRI scripts are not 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. 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.

`ALIGN secname=expression`

Align the section called `secname` to `expression`. The `expression` should be a power of two.

`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`

Does nothing; it’s accepted for compatibility.

`FORMAT output-format`

Similar to the `OUTPUT_FORMAT` command in the more general linker language, but restricted to one of the following output formats:

- S-records, if `output-format` is `S`
- IEEE, if `output-format` is `IEEE`
- 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 `-o` command line option 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 *sec-name*. If you have more than one `SECT` statement for the same *sec-name*, only the *first* sets the start address.

Using make

Copyright © 1991-2000 Free Software Foundation, Inc.

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

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the “GNU General Public License” 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 documentation into another language, under the above conditions for modified versions, except that the documentation 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. For more details, see “General Licenses and Terms for Using GNUPro Toolkit” on page 105 in *Getting Started Guide*.

Free Software Foundation

59 Temple Place / Suite 330
Boston, MA 02111-1307 USA

ISBN: 1-882114-66-3

This documentation has been prepared by Red Hat.

Copyright © 1992-2000 Red Hat

All rights reserved.

1

Overview of `make`, a Program for Recompiling

The `make` utility automatically determines which pieces of a large program need to be recompiled, and then issues commands to recompile them. The following documentation summarizes the `make` utility.

- “Summary of `make` Options” on page 167
- “GNU `make` Quick Reference” on page 219

In the following discussions, the first few paragraphs contain introductory or general information while the subsequent paragraphs contain specialized or technical information; the exception is “Introduction to Makefiles” on page 79, all of which is overview.

If you are familiar with other `make` programs, see “Summary of the Features for the GNU `make` utility” on page 197, “Special Built-in Target Names” on page 104 and “GNU `make`’s Incompatibilities and Missing Features” on page 201 (which explains

the few things GNU `make` lacks that other programs provide).

- “Introduction to Makefiles” on page 79
- “Writing Makefiles” on page 87
- “Writing Rules” on page 93
- “Writing the Commands in Rules” on page 113
- “How to Use Variables” on page 127
- “Conditional Parts of Makefiles” on page 141
- “Functions for Transforming Text” on page 147
- “How to Run the `make` Tool” on page 159
- “Summary of `make` Options” on page 167
- “Implicit Rules” on page 173
- “Using `make` to Update Archive Files” on page 193
- “Summary of the Features for the GNU `make` utility” on page 197
- “GNU `make`’s Incompatibilities and Missing Features” on page 201
- “Makefile Conventions” on page 205
- “GNU `make` Quick Reference” on page 219
- “Complex Makefile Example” on page 227

2

Introduction to Makefiles

`make` is a program that was implemented by Richard Stallman and Roland McGrath. GNU `make` conforms to section 6.2 of *IEEE Standard 1003.2-1992* (POSIX.2). The examples in the documentation for `make` show C programs, since they are most common, although 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. The following documentation discusses the fundamentals of `make`. See also “Writing Makefiles” on page 87.

- “Makefile Rule’s Form” on page 80
- “A Simple Makefile” on page 81
- “How `make` Processes a Makefile” on page 82
- “Variables Make Makefiles Simpler” on page 83

- “Letting make Deduce the Commands” on page 84
- “Another Style of Makefile” on page 85
- “Rules for Cleaning the Directory” on page 86

To prepare to use `make`, you must write a file called the *makefile*, which 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, the following shell command suffices to perform all necessary recompilations.

```
make
```

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 how and which files should be recompiled. If you are new to `make`, or are looking for a general introduction, read the following discussions. 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 the following discussions, we will describe a simple makefile that tells 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 “Complex Makefile Example” on page 227.

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.

Makefile Rule’s Form

A simple makefile consists of *rules* with the following form:

```
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 “Phony Targets” on page 101).

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.

IMPORTANT! You need to have a tabulation 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 “Writing Rules” on page 93. 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.

A Simple Makefile

What follows 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 the following 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`.

IMPORTANT! We split each long line into two lines using backslash-newline (`\`) for paper printing purposes.

```
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
```

To use the previous sample makefile to create the executable `edit` file, use the input, `make`, at the command prompt.

To use this makefile to delete the executable file and all the object files from the directory, use the input, `make clean`, at the command prompt.

The example makefile's 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 first be updated. In the previous 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.

`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. *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 “Phony Targets” on page 101 for information about this kind of target. See “Errors in Commands” on page 117 to see how to cause `make` to ignore errors from `rm` or any other command.

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 “Arguments to Specify the Goals” on page 160.)

See the example shown with “A Simple Makefile” on page 81; its default goal is to update the executable program, `edit`; therefore, we put that rule first.

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; in this case, they 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.

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` along with `command.o` and `files.o`, and then link the file, `edit`.

Variables Make Makefiles Simpler

See the first example with “A Simple Makefile” on page 81; see the list where all the object files repeat twice in the rule for `edit` as in this next example.

```
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 “How to Use Variables” on page 127).

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 input like the following example shows 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)`. The following example shows 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)
```

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 “Implicit Rules” on page 173.

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. The following is the entire example, with both of these changes, and a variable, `objects` (as previously suggested with “Variables

Make Makefiles Simpler” on page 83).

```
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)
```

The example in “Another Style of Makefile” on page 85 is how we would write the makefile in actual practice. (The complications associated with `clean` are described in “Phony Targets” on page 101 and in “Errors in Commands” on page 117.)

Because implicit rules are so convenient, they are used frequently.

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.

The following example shows what such a makefile resembles. `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.

```
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
```

Whether this makefile 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.

Rules for Cleaning the Directory

Compiling a program is not the only thing for which you might want to write rules. Makefiles commonly tell how to do a few other things besides compiling a program; for instance, how to delete all the object files and executables so that the directory is clean. The following shows how to write a `make` rule for cleaning the example editor.

```
clean:
    rm edit $(objects)
```

In practice, you might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. Use input like the following example.

```
.PHONY : clean
clean :
    -rm edit $(objects)
```

This prevents `make` from using an actual file called `clean` allowing it to continue in spite of errors from `rm`. See “Phony Targets” on page 101 and in “Errors in Commands” on page 117. A rule such as this should not be placed at the beginning of the makefile, since you do not want it to run by default! Thus, in the example makefile, you 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, use `make clean`; see also “How to Run the make Tool” on page 159.

3

Writing Makefiles

The information that tells `make` how to recompile a system comes from reading a data base called the *makefile*. The following documentation discusses writing makefiles.

- “What Makefiles Contain” (below)
- “What Name to Give Your Makefile” on page 88
- “Including Other Makefiles” on page 89
- “The MAKEFILES Variable” on page 90
- “How Makefiles are Remade” on page 91
- “Overriding Part of Another Makefile” on page 92

What Makefiles Contain

Makefiles contain five kinds of things: *explicit rules*, *implicit rules*, *variable definitions*, *directives*, and *comments*. Rules, variables, and directives are described in better detail in the corresponding references noted in the following descriptions..

- An *explicit rule* says when and how to remake one or more files called the rule's *targets*. It lists the other files on which the targets depend, and may also give commands to use to create or update the targets. See “Writing Rules” on page 93.
- 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 “Implicit Rules” on page 173.
- 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 “Variables Make Makefiles Simpler” on page 83).
- A *directive* is a command for make to do something special while reading the makefile. These include:
 - Reading another makefile (see “Including Other Makefiles” on page 89).
 - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see “Conditional Parts of Makefiles” on page 141).
 - Defining a variable from a verbatim string containing multiple lines (see “Defining Variables Verbatim” on page 137).
- A *comment* in a line of a makefile starts with #. 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.

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 “Using Implicit Rules” on page 174.

If you want to use a non-standard name for your `makefile`, you can specify the `makefile` name with the `-f` or `--file` option.

The `-f name` or `--file=name` arguments 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`.

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 `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 “How to Use Variables” on page 127. For example, if you have three `.mk` files, `a.mk`, `b.mk`, and `c.mk`, and `$(bar)` expands to `bish bash`, then the 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 “Setting Variables” on page 135) or pattern rules (see “Defining and Redefining Pattern Rules” on page 182). 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 “Generating Dependencies Automatically” on page 109. 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 the `--include-dir` options are searched (see “Summary of make Options” on page 167). Then the directories (if they exist) are searched, in the following 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 do not exist. See “How Makefiles are Remade” on page 91. 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`, as in `-include filenames...`. This acts like `include` in every way except that there is no error (not even a warning) if any of the `filenames` do not exist.

The MAKEFILES Variable

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 in that various directories are searched for those files (see “Including Other Makefiles” on page 89). 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 “Recursive Use of the make Tool” on page 119). 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 “Directory Search and Implicit Rules” on page 101).

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 “Including Other Makefiles” on page 89.

[†] `make` compiled for Microsoft Windows behaves as if `prefix` has been defined to be the root of the Cygwin tree hierarchy.

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 “Using Implicit Rules” on page 174, 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 “Double-colon Rules” on page 109). In the case of makefiles, a make-file 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 “What Name to Give Your Makefile” on page 88. 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 “What Name to Give Your Makefile” on page 88) 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 “Instead of Executing the Commands” on page 162), 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. 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 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`.

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 “Defining and Redefining Pattern Rules” on page 182 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 `makefile` that contains the following content.

```
foo:
    frobnicate > foo

%: force
    @$(MAKE) -f Makefile $@

force: ;
```

If you say `make foo`, `make` will find `makefile`, 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 `makefile`, 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; likewise for any other target that `makefile` 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 following documentation discusses the general rules for makefiles.

- “Rule Syntax” on page 94
- “Using Wildcard Characters in File Names” on page 95
- “Pitfalls of Using Wildcards” on page 96
- “The wildcard Function” on page 96
- “Searching Directories for Dependencies” on page 97
- “Phony Targets” on page 101
- “Rules Without Commands or Dependencies” on page 103

- “Empty Target Files to Record Events” on page 103
- “Special Built-in Target Names” on page 104
- “Multiple Targets in a Rule” on page 105
- “Multiple Rules for One Target” on page 106
- “Static Pattern Rules” on page 107
- “Double-colon Rules” on page 109
- “Generating Dependencies Automatically” on page 109

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 “Defining and Redefining Pattern Rules” on page 182.

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 `a11`). See “Arguments to Specify the Goals” on page 160.

Rule Syntax

In general, a rule looks like the following.

```
targets : dependencies
command
...
```

Or like the following.

```
targets : dependencies ; command
command
...
```

The *targets* are file names, separated by spaces. Wildcard characters may be used (see “Using Wildcard Characters in File Names” on page 95) and a name of the form `a(m)` represents member `m` in archive file `a` (see “Archive Members as Targets” on page 194). Usually there is only one target per rule, but occasionally there is a reason to have more (see “Multiple Targets in a Rule” on page 105). 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 “Writing the Commands in Rules” on page 113.

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 “How to Use Variables” on page 127). 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 “Using `make` to Update Archive Files” on page 193) are allowed 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 “Writing the Commands in Rules” on page 113).

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`. On systems which do not have a home directory for each user (such as Microsoft Windows), this functionality can be simulated by setting the environment variable, `HOME`.

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`.

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 “Empty Target Files to Record Events” on page 103. (The `?$` automatic variable is used to print only those files that have changed; see “Automatic Variables” on page 184.) Wildcard expansion does not happen when you define a variable. Thus, if you write `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 “The wildcard Function” on page 96.

Pitfalls of Using Wildcards

The next 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 the following.

```
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 with “The wildcard Function” on page 96.

Microsoft Windows operating systems use backslashes to separate directories in pathnames (as in `c:\foo\bar\baz.c`; this is equivalent to the Unix-style, `c:/foo/bar/baz.c`, where the `c:` part is the drive letter for the pathname). When `make` runs on these systems, it supports backslashes as well as the Unix-style forward slashes in pathnames. However, this support does not include the wildcard expansion, where backslash is a *quote* character. Therefore, you must use Unix-style slashes in such cases.

The wildcard Function

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, using: `$(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 “Pitfalls of Using Wildcards” on page 96).

One use of the wildcard function is to get a list of all the C source files in a directory, using: `$(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, using the following.

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

Here we have used another function, `patsubst`. See “Functions for String Substitution and Analysis” on page 148.

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 “The Two Flavors of Variables” on page 129 for an explanation of `:=`, which is a variant of `=.`

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. See the following documentation for more specific discussion.

- `VPATH`: Search Path for All Dependencies (this page)
- “The `vpath` Directive” on page 98
- “How Directory Searches Work” on page 99
- “Writing Shell Commands with Directory Search” on page 100
- “Directory Search and Implicit Rules” on page 101
- “Directory Search for Link Libraries” on page 101

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 “Writing Shell Commands with Directory Search” on page 100.

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. (On MS-DOS and MS-Windows, semi-colons are used as separators of directory names in `VPATH`, since the colon can be used in the pathname itself, after the drive letter.)

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 rule, `foo.o : foo.c`, is interpreted as if it were written: `foo.o : src/foo.c`, assuming the file, `foo.c`, does not exist in the current directory but is found in the `src` directory.

The `vpath` Directive

Similar to the `VPATH` variable but more selective is the `vpath` directive (note the use of 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 (on MS-DOS and MS-Windows, semi-colons are used) 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 “Defining and Redefining Pattern Rules” on page 182). 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, the following directive will look for a file ending in `.c` in `foo`, then `blish`, then `bar`.

```
vpath %.c foo
vpath % blish
vpath %.c bar
```

The next example, on the other hand, will look for a file ending in `.c` in `foo`, then `bar`, then `blish`.

```
vpath %.c foo:bar
vpath % blish
```

How Directory Searches Work

When a dependency is found through directory search, regardless of type (general or selective), the pathname located may not be the one that `make` actually provides you in the dependency list. Sometimes the path discovered through directory search is thrown away. The algorithm `make` uses to decide whether to keep or abandon a path found during a directory search has the following method.

1. If a target file does not exist at the path specified in the makefile, directory search is performed.
2. If the directory search is successful, that path is kept and the file is tentatively stored as the target.
3. All dependencies of the target are examined using this same method.
4. After processing the dependencies, the target may or may not need to be rebuilt, depending on the following circumstances:

- If the target does not need to be rebuilt, the path to the file found during directory search is used for any dependency lists containing the target. In short, if `make` does not need to rebuild the target, you use the path found during a directory search.
- If the target does need to be rebuilt (being obsolete), the pathname found during a directory search is unused, and the target is rebuilt using the file name specified in the makefile.

In short, if `make` must rebuild, then the target is rebuilt locally, not in the directory found during a directory search.

This algorithm may seem complex; in practice, it is quite often exactly what you want.

Other versions of `make` use a simpler algorithm; if the file does not exist, and it is found during a directory search, then that pathname is always used whether or not the target needs to be built. Thus, if the target is rebuilt it is created at the pathname discovered during directory search.

If, in fact, this is the behavior you want for some or all of your directories, you can use the `GPATH` variable to indicate this to `make`. `GPATH` has the same syntax and format as `VPATH` (that is, a space- or colon-delimited list of pathnames). If an obsolete target is found by directory search in a directory that also appears in `GPATH`, then that pathname is not thrown away. The target is rebuilt using the expanded path.

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 `$$` (see “Automatic Variables” on page 184). For instance, the value of `$$` 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, as in the following example.

```
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 “Variables Used by Implicit Rules” on page 179.

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, as in the following declaration.

```
VPATH = src:../headers
foo.o : foo.c defs.h
hack.h cc -c $(CFLAGS) $$< -o $$
```

Directory Search and Implicit Rules

The search through the directories specified in `VPATH` or with `vpath` also happens during consideration of implicit rules (see “Using Implicit Rules” on page 174). 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.

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, `-lname`. (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 `-lname`.) When a dependency’s name has the form `-lname`, `make` handles it specially by searching for the file `libname.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`). Use the following example, for instance.

```
foo : foo.c -lcurses
    cc $^ -o $@
```

This 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`.

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. Use the following, for 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 use `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 “Special Built-in Target Names” on page 104), as in: `.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 “Implicit Rules” on page 173). 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 the following example.

```
.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 “Arguments to Specify the Goals” on page 160).

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. Use the following, 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 use 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 instance, in the following example, `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
```

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. The following example will illustrate the rule.

```
clean: FORCE
    rm $(objects)
FORCE:
```

In this case, 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 “Phony Targets” on page 101.

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. Use the following as 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 “Automatic Variables” on page 184).

Special Built-in Target Names

The following 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 “Phony Targets” on page 101.

`.SUFFIXES`

The dependencies of the special target, `.SUFFIXES`, are the list of suffixes to be used in checking for suffix rules. See “Old-fashioned Suffix Rules” on page 189.

`.DEFAULT`

The commands specified for `.DEFAULT` are used for any target for which no rules are found (either explicit rules or implicit rules). See “Defining Last-resort Default Rules” on page 188. 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 “Implicit Rule Search Algorithm” on page 191.

`.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 “Interrupting or Killing the make Tool” on page 118. Also, if the target is an intermediate file, it will not be deleted after it is no longer needed, as is normally done. See “Chains of Implicit Rules” on page 181.

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 “Errors in Commands” on page 117.

`.SILENT`

If you specify dependencies for `.SILENT`, then `make` will not 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 “Command Echoing” on page 114.

If you want to silence all commands for a particular run of `make`, use the `-s` or `--silent` options. See “Summary of make Options” on page 167.

`.EXPORT_ALL_VARIABLES`

Simply by being mentioned as a target, this tells `make` to export all variables to child processes by default.

See “Communicating Variables to a Sub-make Utility” on page 120.

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 “Old-fashioned Suffix Rules” on page 189.

Multiple Targets in a Rule

A rule with multiple targets is the same as writing many rules, each with one target, and all identical aside from that issue. The same commands apply to all the targets, although their effects vary, since you substitute an actual target name into the command (using `$$`). The rule also contributes the same dependencies to all targets, useful in two cases.

- You want just dependencies, no commands. Use the following for an example.

```
kbd.o command.o files.o: command.h
```

This input 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 “Automatic Variables” on page 184). Use the following for an example.

```
bigoutput littleoutput : text.g
    generate text.g -$(subst output,,$$) > $$
```

This input is equivalent to the next example.

```
bigoutput : text.g
    generate text.g -big > bigoutput
littleoutput : text.g
    generate text.g -little > littleoutput
```

The hypothetical program, `generate`, makes two types of output, one if given `-big` and one if given `-little`. See “Functions for String Substitution and

Analysis” on page 148 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 “Static Pattern Rules” on page 107.

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 input.

```
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 problem is that the additional dependencies could be specified with a variable that you set with a command argument to `make` (see “Overriding Variables” on page 164). Use the following for an example.

```
extradeps=
$(objects) : $(extradeps)
```

This input 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 “Using Implicit Rules” on page 174).

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*.

Syntax of Static Pattern Rules

The following shows the syntax of a static pattern rule:

```
targets ...: target-pattern: dep-patterns ...
      commands
      ...
```

The *targets* list specifies the targets to which the rule applies. The targets can contain wildcard characters, just like the targets of ordinary rules (see “Using Wildcard Characters in File Names” on page 95).

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. The following is an example which compiles each of `foo.o` and `bar.o` from the corresponding `.c` file.

```
objects = foo.o bar.o

all: $(objects)
```

```
$(objects): %.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@
```

In the previous example, `$<` is the automatic variable that holds the name of the dependency and `$@` is the automatic variable that holds the name of the target; see “Automatic Variables” on page 184. 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 “Functions for String Substitution and Analysis” on page 148), as in the following example.

```
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`.

The following 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`.

Static Pattern Rules Compared to Implicit Rules

A static pattern rule has much in common with an implicit rule defined as a pattern rule (see “Defining and Redefining Pattern Rules” on page 182). 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 is an error. The static pattern rule can be better than an implicit rule for the following 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.

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.

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 “Using Implicit Rules” on page 174.

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` using 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`. You no longer have to write all those rules yourself. The compiler will do it for you.

IMPORTANT! 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` will not ever remove the file after using it; see “Chains of Implicit Rules” on page 181.

With old `make` programs, it was common 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 “Including Other Makefiles” on page 89).

In `make`, the feature of remaking makefiles makes this practice obsolete; you need never tell `make` explicitly to regenerate dependencies, because it always regenerates any makefile that is out of date. See “How Makefiles are Remade” on page 91.

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`, listing which files on which the object file, `name.o`, depends. That way only the source files that have changed need to be rescanned to produce the new dependencies.

The following is an example of the pattern rule to generate a file of dependencies (a makefile) called `name.d` from a C source file called `name.c`.

```
%.d: %.c
    $(SHELL) -ec '$(CC) -M $(CPPFLAGS) $< \
    | sed '\''s/$*\.\.o[ :]*/& $@/g'\'' > $@'
    [ -s $@ ] || rm -f $@'
```

See “Defining and Redefining Pattern Rules” on page 182 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 “Options Controlling the Preprocessor” in *Using GNU CC* in **GNUPro Compiler Tools** for details. For example, the purpose of the `sed` command is to translate `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 on which the corresponding `.o` file depends. `make` then knows it must regenerate the dependencies whenever any of the source or header files changes. Once you have defined the rule to remake the `.d` files, you then use the `include` directive to read them all in. See “Including Other Makefiles” on page 89. Use the following example, for clarification.

```
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 “Substitution References” on page 131 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 “How Makefiles are Remade” on page 91.

5

Writing the Commands in Rules

The commands of a rule consist of shell command lines to be executed one by one. The following documentation discusses this execution of shell commands.

- “Command Echoing” on page 114
- “Command Execution” on page 114
- “Parallel Execution” on page 116
- “Errors in Commands” on page 117
- “Interrupting or Killing the make Tool” on page 118
- “Recursive Use of the make Tool” on page 119
- “Defining Canned Command Sequences” on page 124
- “Using Empty Commands” on page 125

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.

WARNING! An apparently “blank” line that begins with a tab is *not* blank. It is an empty command; see “Using Empty Commands” on page 125.)

Users use many different shell programs, but commands in makefiles are always interpreted by `/bin/sh` unless the makefile specifies otherwise. See “Command Execution” on page 114. 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.

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 `-nor --just-print`, echoing is all that happens with no execution. See “Summary of make Options” on page 167. 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 “Special Built-in Target Names” on page 104). `.SILENT` is essentially obsolete since `@` is more flexible.

Command Execution

When executing commands to update a target, `make` a new subshell for each line. (In practice, `make` may take shortcuts that do not affect the results.)

IMPORTANT! The implication that shell commands such as `cd` 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. Use the following example for clarification.[†]

```
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.

For Windows, if `SHELL` is not set, the value of the `COMSPEC` variable (which is always set) is used instead.

The processing of lines that set the variable `SHELL` in Makefiles is different for Windows. The stock shell, `command.com`, is ridiculously limited in its functionality and many users of `make` tend to install a replacement shell. Therefore, `make` examines the value of `SHELL`, and changes its behavior based on whether it points to a Unix-style or Windows-style shell. This allows reasonable functionality even if `SHELL` points to `command.com`; if `SHELL` points to a Unix-style shell, `make` for Windows additionally checks whether that shell can indeed be found; if not, it ignores the line that sets `SHELL`. For Windows, `make` searches for the shell in the following places.

- In the precise place pointed to by the value of `SHELL`. If the makefile specifies `SHELL = /bin/sh`, `make` will look in the `bin` directory on the current drive.
- In the current directory.
- In each of the directories in the `PATH` variable, in order.

In every directory it examines, `make` will first look for the specific file (`sh` in the previous example). If this is not found, it will also look in the directory for that file with one of the known extensions identifying executable files (for example, `.exe`, `.com`, `.bat`, `.btm`, `.sh`, and some others).

If any of these attempts is successful, the value of `SHELL` will be set to the full pathname of the shell as found. However, if none of these is found, the value of `SHELL` will not be changed, and thus the line setting it will be effectively ignored. This is so `make` will only support features specific to a Unix-style shell *if* such a shell is actually installed on the system where `make` runs.

This extended search for the shell is limited to the cases where `SHELL` is set from the Makefile; if it is set in the environment or command line, you are expected to set it to the full pathname of the shell, exactly as things are on Unix.

[†] For Windows, the value of current working directory is `global`, so changing that value *will* affect the command lines following such commands on Windows systems.

The effect of the Windows-specific processing is that a Makefile, having the input of `SHELL = /bin/sh`(as many Unix makefiles do), will work for Windows unaltered, *if* you have a file such as `sh.exe` installed in some directory along that `PATH`.

Unlike most variables, `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 “Variables from the Environment” on page 138.

However, for Windows, the value of `SHELL` in the environment is used, since on those systems most users do not set this variable, and therefore it is most likely set specifically to be used by `make`. For Windows, if the setting of `SHELL` is not suitable for `make`, you can set the variable, `MAKESHELL` to the shell that `make` should use; this will override the value of `SHELL`.

Parallel Execution

`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 `-jor --jobs` option tells `make` to execute many commands simultaneously.

For Windows, the `-j` option has no effect, since that system doesn’t support multi-processing.

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 (for instance, if it is killed by a signal or exits with a nonzero status), and errors are not ignored for that command (see “Errors in Commands” on page 117), 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 “Summary of make Options” on page 167), `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.

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, as in the following 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` or the `-iflag`, `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 “Summary of `make` Options” on page 167.

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 `Emacscompile` 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 “Interrupting or Killing the `make` Tool” on page 118. 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.

Interrupting or Killing the `make` Tool

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 use `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.

Recursive Use of the `make` Tool

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 the following

```
subsystem:
    cd subdir; $(MAKE)
```

Or, equivalently (see “Summary of `make` Options” on page 167), use the following input.

```
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`.

For your convenience, `make` sets the `CURDIR` variable to the pathname of the current working directory for you. If `-C` is in effect, it will contain the path of the new directory, not the original. The value has the same precedence it would have if it were set in the makefile (by default, an environment variable, `CURDIR`, will not override this value). Setting this variable has no effect on the operation of `make`.

How the `MAKE` Variable Works

Recursive `make` commands should always use the variable, `MAKE`, not the explicit command name, `make`, as the following example shows.

```
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)` options. Using the `MAKE` variable has the same effect as using a `+` character at the beginning of the command line. See “Instead of Executing the Commands” on page 162.

Consider the command `make -t` for example. The `-t` option marks targets as up to date without actually running any commands; see “Instead of Executing the Commands” on page 162. Following the usual definition of `-t`, a `make -t` command would create a file named `subsystem` and do nothing else. What you really want it to do is run `cd subdir; make -t` although 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 `-t`, `-n` and `-q` flags 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 “Communicating Options to a Sub-make Utility” on page 122), so your request to touch the files, or print the commands, is propagated to the subsystem.

Communicating Variables to a Sub-make Utility

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 `-eswitch` (see “Summary of make Options” on page 167).

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 “Variables from the Environment” on page 138. 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 “Communicating Options to a Sub-make Utility” on page 122.

Variables are *not* normally passed down if they were created by default by `make` (see “Variables Used by Implicit Rules” on page 179). The sub-`make` will define these for

itself.

If you want to export specific variables to a sub-`make`, use the `export` directive like:

```
export variable ....
```

If you want to prevent a variable from being exported, use the `unexport` directive, like:

```
unexport variable ....
```

As a convenience, you can define a variable and export it at the same time by using `export variable = value` (which has the same result as `variable = value export variable`) and `export variable := value` (which has the same result as `variable := value export variable`).

Likewise, `export variable += value` is just like `variable += value export variable`.

See “Appending More Text to Variables” on page 135. 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 alphanumeric 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 `makes`, 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 “Conditional Parts of Makefiles” on page 141); 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 “The `MAKEFILES` Variable” on page 90.

Communicating Options to a Sub-make Utility

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 “Summary of make Options” on page 167.

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 “Overriding Variables” on page 164.

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 “Parallel Execution” on page 116). 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 was passed down to sub-makes, 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 the following example shows.

```
MAKEFLAGS=
subsystem:
    cd subdir; $(MAKE)
```

Alternately, use the following example’s input.

```
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 `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 the following.

```
subsystem:
    cd subdir; $(MAKE) $(MFLAGS)
```

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 “Summary of `make` Options” on page 167), 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.

IMPORTANT! 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.

The `--print-directory` Option

If you use several levels of recursive `make` invocations, the options, `-w` or `--print-directory` 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 like the following before doing anything else.

```
make: Entering directory /u/gnu/make.
```

Then, a line of the following form when processing is completed.

```
make: Leaving directory /u/gnu/make.
```

Normally, you do not need to specify this option because `make` does it for you: `-w` is

turned on automatically when you use the `-coption`, 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.

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.

The following is an example of defining a canned sequence of commands.

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

`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 “Defining Variables Verbatim” on page 137 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 “Basics of Variable References” on page 128). Because variables defined by `define` are recursively expanded variables, all the variable references you wrote inside the `define` are expanded now. Use the following 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 “Using Implicit Rules” on page 174).

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 “Summary of `make` Options” on page 167.

For example, use the following example of a canned sequence.

```
define frobnicate
@echo "frobnicating target $@"
frob-step-1 $< -o $@-step-1
frob-step-2 $@-step-1 -o $@
endif
```

`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 following rule statement does not echo any commands. (See “Command Echoing” on page Command Echoing for a full explanation of `@`.)

```
frob.out: frob.in
@$(frobnicate)
```

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. The only reason this is useful is to prevent a target from getting implicit commands (from implicit rules or the `.DEFAULT` special target; see “Implicit Rules” on page 173 and “Defining Last-resort Default Rules” on page 188).

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 “Phony Targets” on page 101 for a better way to execute this requirement.

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*. The following documentation discusses using variables.

- “Basics of Variable References” on page 128
- “The Two Flavors of Variables” on page 129
- “How Variables Get Their Values” on page 134
- “Setting Variables” on page 135
- “Appending More Text to Variables” on page 135
- “The override Directive” on page 137
- “Defining Variables Verbatim” on page 137
- “Variables from the Environment” on page 138
- “Target-specific Variable Values” on page 139

- “Pattern-specific Variable Values” on page 140

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 because 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 “Communicating Variables to a Sub-`make` Utility” on page 120).

Variable names are case-sensitive. The names, `foo`, `FOO`, and `Foo`, all refer to different variables. It is traditional to use uppercase letters in variable names, but we recommend using lowercase letters for variable names that serve internal purposes in the makefile, and reserving uppercase for parameters that control implicit rules or for parameters that the user should override with command options (see “Overriding Variables” on page 164).

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 “Automatic Variables” on page 184.

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. The following 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 following rule could be used to compile a C program `prog.c`.

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

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 “Automatic Variables” on page 184).

The Two Flavors of Variables

There are two ways that a variable in `make` can have a value; we call them the two *flavors* of variables.

- ***Recursively expanded variables***
- ***Simply expanded variables***

The two flavors are differentiated in how they are defined and in what they do when expanded. The following documentation discusses the distinctions.

Recursively expanded variables are defined by lines using `=` (see “Setting Variables” on page 135) or by the `define` directive (see “Defining Variables Verbatim” on page 137). 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*. Consider the following example.

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?
```

```
all:;echo $(foo)
```

This input 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 the following statement will do what was intended: when `CFLAGS` is expanded in a command, it will expand to `-Ifoo -Ibar -O`.

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

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 referenced in the definition will be

executed every time the variable is expanded (see “Functions for Transforming Text” on page 147). 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 “Setting Variables” on page 135). 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, consider the following statement.

```
x :=foo
y := $(x) bar
x := later
```

The previous input is equivalent to the next statement.

```
y := foo bar
x := later
```

When a simply expanded variable is referenced, its value is substituted verbatim. The following is a somewhat more complicated example, illustrating the use of `:=in` conjunction with the shell function. See “The shell Function” on page 156. This example also shows use of the variable, `MAKELEVEL`, which is changed when it is passed down from level to level. See “Communicating Variables to a Sub-make Utility” on page 120 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 “Functions for Transforming Text” on page 147).

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 the following example's input.

```
nullstring :=
space := $(nullstring) # end of the line
```

With this statement, 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 the following.

```
dir := /foo/bar          # directory to put the frobs in
```

With this statement, 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!)

There is another assignment operator for variables, `?=`, called a *conditional variable assignment operator*, because it only has an effect if the variable is not yet defined.

The statement, `FOO ?= bar`, has exactly the equivalent definition as in the following example's input (see also “The origin Function” on page 155).

```
ifeq ($(origin FOO), undefined)
    FOO = bar
endif
```

A variable set to an empty value is still defined, so `?=` will not set that variable.

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. The following input sets `bar` to `a.c b.c c.c..`

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

See “Setting Variables” on page 135. A substitution reference is actually an abbreviation for use of the `patsubst` expansion function; see “Functions for String Substitution and Analysis” on page 148. 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 “Functions for String Substitution and Analysis” on page 148 for a description of the `patsubst` function. Consider the following example, setting `bar` to `a.c b.c c.c..`

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

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, the following documentation will elucidate the concept of computed variable names.

Variables may be referenced *inside* the name of a variable. This is called a *computed variable name* or a *nested variable reference*. Consider the next example.

```
x =y
y =z
a := $( $(x) )
```

The previous example 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)`, is nested within the outer variable reference. The previous example shows two levels of nesting; however, any number of levels is possible. For instance, the following example shows three levels.

```
x =y
y =z
z =u
a := $( $( $(x) ) )
```

The previous example shows 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. Consider the following example.

```
x = $(y)
y =z
z = Hello
a := $( $(x) )
```

The previous example shows `a` defined as `Hello`; `$($(x))` becomes `$($(y))` which becomes `$(z)` which becomes `Hello`.

Nested variable references can also contain modified references and function invocations (see “Functions for Transforming Text” on page 147), just like any other

reference. For instance, the following example uses the `subst` function (see “Functions for String Substitution and Analysis” on page 148):

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z =y a := $( $(z) )
```

The previous example 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. Consider the following example.

```
a_dirs := dira dirb
l_dirs := dir1 dir2

a_files := filea fileb
l_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) )
```

The previous example will give `dirs` the same value as `a_dirs`, `l_dirs`, `a_files` or `l_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
l_objects := 1.o 2.o 3.o

sources := $( $(a1)_objects:.o=.c )
```

The previous example defines `sources` as `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 as in the following example.

```
ifdef do_sort
func := sort
else
func := strip
endif

bar:=a d b g q c

foo := $($func) $(bar)
```

The previous example 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 `ad b gq 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 the following example.

```
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`.

IMPORTANT! *Nested variable references* are quite different from *recursively expanded variables* (see “The Two Flavors of Variables” on page 129), though both are used together in complex ways when doing makefile programming.

How Variables Get Their Values

Variables can get values in several different ways:

- You can specify a value in the makefile, either with an assignment (see “Setting Variables” on page 135) or with a verbatim definition (see “Defining Variables Verbatim” on page 137).
- Variables in the environment become make variables. See “Variables from the Environment” on page 138.
- You can specify an overriding value when you run `make`. See “Overriding Variables” on page 164.
- Several variables have constant initial values. See “Variables Used by Implicit Rules” on page 179.
- Several automatic variables are given new values for each rule. Each of these has a single conventional use. See “Automatic Variables” on page 184.

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 “The Two Flavors of Variables” on page 129.

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 “Variables Used by Implicit Rules” on page 179). Several special variables are set automatically to a new value for each rule; these are called the *automatic variables* (see “Automatic Variables” on page 184). If you’d like a variable to be set to a value only if it’s not already set, then you can use the `?=` shorthand operator instead of `=` as the following two settings show, where the `FOO` variables are identical (see also “The origin Function” on page 155):

```
FOO ?= bar
and
ifeq ($(origin FOO), undefined)
FOO = bar
endif
```

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 `+=`, as in `objects += another.o`.

This takes the value of the variable `objects`, and adds the text `another.o` to it (preceded by a single space), as in the following example.

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

The last line sets `objects` to `main.o foo.o bar.o.o an utils other.o`.

Using += is similar to the following example.

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

This last example’s statement 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 =, defining it as a recursively-expanded variable. However, when there is a previous definition, exactly what += does depends on what flavor of variable you defined originally. See “The Two Flavors of Variables” on page 129 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 “Setting Variables” on page 135 for a full explanation of :=). Consider the following definition.

```
variable := value
variable += more
```

This last statement is exactly equivalent to this next definition.

```
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 “The Two Flavors of Variables” on page 129).

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
```

This last statement is roughly equivalent to this next definition.

```
temp = value
variable = $(temp) more
```

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 “Catalogue of Implicit Rules” on page 175.) 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 with this following definition.

```
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 that, if that variable gets defined at any later point, a reference like `$(CFLAGS)` still uses its value.

The override Directive

If a variable has been set with a command argument (see “Overriding Variables” on page 164), 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 looking like `override variable = value`, or `override variable := value`.

To append more text to a variable defined on the command line, use the statement, `override variable += more text`.

See “Appending More Text to Variables” on page 135. 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 the `override CFLAGS += -g` `override` directive. You can also use `override` directives with `define` directives. This is done as you might expect, as in the following statement.

```
override define foo
bar
endif
```

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 “Defining

Canned Command Sequences” on page 124).

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, `undef`. Aside from this difference in syntax, `define` works just like `=`; it creates a recursively-expanded variable (see “The Two Flavors of Variables” on page 129). 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)
undef
```

The value in an ordinary assignment cannot contain a newline; 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 `undef` and is not considered part of the value). The previous example is functionally equivalent to `two-lines = echo foo; echo $(bar)` since two commands separated by semicolon behave much like two separate shell commands. However, using two separate lines means `make` will invoke the shell twice, running an independent sub-shell for each line. See “Command Execution” on page 114. 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` as in the following example.

```
override define two-lines
foo
$(bar)
undef
```

See “The override Directive” on page 137.

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 `-eflag` is specified, then values from the environment override assignments in the makefile (see “Summary of `make` Options” on page 167), although this is not recommended practice.)

Thus, by setting the `CFLAGS` variable 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. However, 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 “Recursive Use of the `make` Tool” on page 119). 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 “Communicating Variables to a Sub-`make` Utility” on page 120 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`. (except for Windows, where `SHELL` is usually not set; see also “Command Execution” on page 114.)

Target-specific Variable Values

Variable values in `make` are usually *global*; that is, they are the same regardless of where they are evaluated (unless they are reset, of course). One exception to that is *automatic variables* (see “Automatic Variables” on page 184).

The other exception is target-specific variable values. This feature allows you to define different values for the same variable, based on the target that `make` is currently building. As with automatic variables, these values are only available within the context of a target’s command script (and in other target-specific assignments).

Set a target-specific variable value, using the following input example’s form.

```
target ... : variable-assignment
```

Alternatively, use the following example’s input form.

```
target ... : override variable-assignment
```

Multiple target values create a target-specific variable value for each member of the target list individually. The variable-assignment can be any valid form of assignment; *recursive* (`=`), *static* (`:=`), *appending* (`+=`), or *conditional* (`?=`). All variables that appear within the variable-assignment are evaluated within the context of the target; thus, any previously-defined target-specific variable values will be in effect. This variable is actually distinct from any global value; the two variables do not have to have the same flavor (recursive *or* static).

Target-specific variables have the same priority as any other makefile variable.

Variables provided on the command-line (and in the environment if the `-e` option is in force) will take precedence. Specifying the `override` directive will allow the target-specific variable value to have precedence.

There is one more special feature of target-specific variables: when you define a target-specific variable, that variable value is also in effect for all dependencies of this target (unless those dependencies override it with their own target-specific variable value). So, for example, input like the following example shows would will set `CFLAGS` to `-g` in the command script for `prog` and it will also set `CFLAGS` to `-g` in the command scripts that create `prog.o`, `foo.o`, and `bar.o`, and any command scripts creating their dependencies.

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o -sp
```

Pattern-specific Variable Values

In addition to target-specific variable values (see “Target-specific Variable Values” on page 139), `make` supports pattern-specific variable values. In this form, a variable is defined for any target that matches the pattern specified. Variables defined in this way are searched after any target-specific variables defined explicitly for that target, and before target-specific variables defined for the parent target.

Set a pattern-specific variable value like the following example input’s form shows.

```
pattern ... : variable-assignment
```

Alternatively, use the following example input’s form.

```
pattern ... : override variable-assignment
```

pattern signifies a %-pattern. As with target-specific variable values, multiple pattern values create a pattern-specific variable value for each pattern, individually. The variable-assignment can be any valid form of assignment. Any command-line variable setting will take precedence, unless `override` is specified. The following example input’s form will assign `CFLAGS` the value of `-O` for all targets matching the `%.o` pattern.

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. The following documentation describes conditionals; see also “Syntax of Conditionals” on page 143 and “Conditionals That Test Flags” on page 145.

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 to use but also which libraries to link.

```
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, specifying 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 the following 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, and 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 previous example has this effect.

```
foo: $(objects)
    $(CC) -o foo $(objects) $(libs_for_gcc)
```

When the variable, `CC`, has any other value, it takes the following effect.

```
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 as in the following example.

```
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)
```

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. The following is a description 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 “Functions for String Substitution and Analysis” on page 148) to avoid interpreting whitespace as a non-empty value. For example, the following will evaluate `text-if-empty` even if the expansion of `$(foo)` contains whitespace characters.

```
ifeq ($(strip $(foo)),)
text-if-empty
endif
```

```
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 differ, the `text-if-true` is effective; otherwise, `text-if-false`, if any, is.

`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.

IMPORTANT! `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),)`, as in the following example.

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

The previous definition example sets `frobozz` to `yes` while the following definition sets `frobozz` to `no`.

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

`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 “Automatic Variables” on page 184). 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.

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 “Functions for String Substitution and Analysis” on page 148). 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, the following shows 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 “Recursive Use of the `make` Tool” on page 119.

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*) on which the function operates. 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.

The following documentation discusses functions in more detail.

- “Functions for String Substitution and Analysis” on page 148
- “Functions for File Names” on page 151
- “The foreach Function” on page 153
- “The origin Function” on page 155
- “The shell Function” on page 156

Function Call Syntax

A function call resembles a variable reference.

It looks like: `$(function arguments)`; or like: `${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 paren-theses 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 the following.

```
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.

Functions for String Substitution and Analysis

The following functions 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 fEEt on the strEEt string.

```
$(patsubst pattern, replacement, text)
```

Finds whitespace-separated words in *text* that match *pattern* and replaces them with *replacement*. In this string, *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 compares 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 are a simpler way to get the effect of the *patsubst* function; see “Substitution References” on page 131.

```
$(var: pattern=replacement)
```

The previous example of a substitution reference is equivalent to the following example’s input.

```
$(patsubst pattern,replacement,$(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)
```

The previous example’s shorthand is equivalent to the following example’s input.

```
$(patsubst %suffix,%replacement,$(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 an empty string using `ifeq` or `ifneq`, you usually want a string of just whitespace to match the empty string (see “Conditional Parts of Makefiles” on page 141).

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 following examples produce, respectively, the values `a` and an empty string.

```
$(findstring a,a b c)
```

```
$(findstring a,b c)
```

See “Conditionals That Test Flags” on page 145 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 words that *do* match. The patterns are written using `%`, just like the patterns used in the `patsubst` function.

The `filter` function can be used to separate out different types of strings (such as file names) in a variable. Consider the following, for example.

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
    cc $(filter %.c %.s,$(sources)) -o foo
```

With this statement, `foo` depends on `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. Consider the following for example.

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

Given the previous lines, the following then 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.

The following 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 “`VPATH`: Search Path for All Dependencies” on page 98). The following 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, as in the following.

```
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 “The `override` Directive” on page 137).

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 resulting file name, `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, sep-arated 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)` results in `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)` results in `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 than 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`.

`$(wordlist s, e, text)`

Returns the list of words in *text* starting with word, *s*, and ending with word, *e* (inclusive). The legitimate values of *s* and *e* start from 1. If *s* is bigger than the number of words in *text*, the value is empty. If *e* is bigger than the number of words in *text*, words up to the end of *text* are returned. If *s* is greater than *e*, make swaps them for you. The input, `$(wordlist 2, 3, foo bar baz)`, returns `bar baz` as a result.

`$(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 “Using Wildcard Characters in File Names” on page 95.

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; 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`.

The following 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)/*))
```

With the previous example, `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/*)`. The previous example has the same result (except for setting `dirs`) as `files:= $(wildcard a/* b/* c/* d/*)`.

When `text` is complicated, you can improve readability by giving it a name, with an additional variable, as in the following.

```
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 re-expanded 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 “The Two Flavors of Variables” on page 129.

You must take care when using complex variable expressions that result in variable names because many strange things are valid variable names, and are probably not what you intended. Consider the following, for example.

```
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
```

This expression might be useful if the value of `find_files` references the variable whose name is `Esta escrito en espanol!`, but it is more likely to be a mistake.

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 its origin. Its syntax is:

```
$(origin variable)
```

IMPORTANT! *variable* is the name of a variable to inquire about; it is *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 as the following descriptions discuss.

undefined

Used if *variable* was never defined.

default

Used if *variable* has a default definition as is usual with `CC` and so on. See “Variables Used by Implicit Rules” on page 179.

IMPORTANT! If you have redefined a default variable, the `origin` function will return the origin of the later definition.

environment

Used if *variable* was defined as an environment variable and the `-e` option is not turned on (see “Summary of make Options” on page 167).

environment override

Used if *variable* was defined as an environment variable and the `-e` option is turned on (see “Summary of make Options” on page 167).

file

Used if *variable* was defined in a makefile.

command line

Used if *variable* was defined on the command line.

override

Used if *variable* was defined with an `override` directive in a makefile (see “The override Directive” on page 137).

automatic

Used if *variable* is an automatic variable defined for the execution of the commands for each rule (see “Automatic Variables” on page 184).

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 `make -f bar` command, 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 the following statement.

```
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 the following statement.

```
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 “Functions for String Substitution and Analysis” on page 148.

The `shell` Function

The `shell` function is unlike any other function except the `wildcard` function (see “The wildcard Function” on page 96) 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 newline or a carriage-return /newline pair to a single space. It also removes the trailing newline (accompanying the carriage-return), if it is the last thing in the result.

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. The following is an example of the use of the `shell` function which sets contents to the contents of the file, `foo`, with a space (rather than a newline) separating each line.

```
contents := $(shell cat foo)
```

The following is an example of the use of the `shell` function which sets files to the expansion of `*.c`. Unless `make` is using a very strange shell, this has the same result as `$(wildcard *.c)`.

```
files := $(shell echo *.c)
```


9

How to Run the `make` Tool

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. However, 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. The following documentation provides more details with running `make` for recompilation.

- “Arguments to Specify the Makefile” on page 160
- “Arguments to Specify the Goals” on page 160
- “Instead of Executing the Commands” on page 162
- “Avoiding Recompilation of Some Files” on page 164
- “Overriding Variables” on page 164
- “Testing the Compilation of a Program” on page 165

Arguments to Specify the Makefile

By giving arguments when you run `make`, you can do many things.

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 “Instead of Executing the Commands” on page 162.

The way to specify the name of the makefile is with the `-f` or `--file` options (`--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 “Writing Makefiles” on page 87).

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. 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.

`make` will set the special variable, `MAKECMDGOALS`, to the list of goals you specified on the command line. If no goals were given on the command-line, this variable is empty.

IMPORTANT! `MAKECMDGOALS` should be used only in special circumstances. The following

example shows the appropriate use in order to avoid including `.d` files during `clean` rules (see also “Generating Dependencies Automatically” on page 109), so `make` won’t create them only to immediately remove them again:

```
sources = foo.c bar.c

ifneq ($(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

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 the following.

```
.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 “Phony Targets” on page 101) or empty target (see “Empty Target Files to Record Events” on page 103). 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 “Standard Targets for Users” on page 207 for a detailed list of all the standard target names which GNU software packages use.

`all`

Makes all the top-level targets about which the makefile knows.

`clean`

Deletes 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`

Copies 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`

Prints listings of the source files that have changed.

`tar`

Creates a `tar` file of the source files.

`shar`

Creates a shell archive (`shar` file) of the source files.

`dist`

Creates 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 previous targets, or even more than one of the previous targets.

`TAGS`

Updates a tags table for this program.

`check`

`test`

Performs self tests on the program this makefile builds.

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.

The following 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 the three flags, `-n`, `-t`, and `-q`, 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}`. 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 “How the MAKE Variable Works” on page 119.

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.

The options, `-p` and `-v`, allow you to obtain other information about `make` or about the makefiles in use (see “Summary of make Options” on page 167).

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.

Use the following 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 “Summary of `make` Options” on page 167), meaning that the file itself will not be remade, and nothing else will be remade on its account.

Use the following 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`.

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 using

something like: `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: `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 “Variables Used by Implicit Rules” on page 179 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 previously make a recursively-expanded variable; to make a simply-expanded variable, write `:=` instead of `=`. 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 using something like `override variable=value` (see “The override Directive” on page 137).

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 “Summary of `make` Options” on page 167).

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 the Emacs **Meta-x** `compile` command passes the `-k` flag by default.

10

Summary of `make` Options

The following documentation discusses the options for `make`.

`-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 “Recursive Use of the `make` Tool” on page 119).

`-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 “Variables from the Environment” on page 138.

`-f file`

`--file=file`

`--makefile=file`

Read the file named *file* as a makefile. See “Writing Makefiles” on page 87.

`-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 “Errors in Commands” on page 117.

`-I dir`

`--include-dir=dir`

Specifies a directory, *dir*, to search for included makefiles. See “Including Other Makefiles” on page 89. 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 “Parallel Execution” on page 116 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 “Testing the Compilation of a Program” on page 165.

`-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 “Parallel Execution” on page 116.

`-n`
`--just-print`
`--dry-run`
`--recon`
Print the commands that would be executed, but do not execute them. See “Instead of Executing the Commands” on page 162.

`-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 “Avoiding Recompilation of Some Files” on page 164.

`-p`
`--print-data-base`
Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as other-wise specified. This also prints the version information given by the `-v` switch (see “`-v`” on page 170). 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 “Instead of Executing the Commands” on page 162.

`-r`
`--no-builtin-rules`
Eliminate use of the built-in implicit rules (see “Using Implicit Rules” on page 174). You can still define your own by writing pattern rules (see “Defining and Redefining Pattern Rules” on page 182). The `-r` option also clears out the default list of suffixes for suffix rules (see “Old-fashioned Suffix Rules” on page 189). But you can still define your own suffixes with a rule for `.SUFFIXES`, and then define your own suffix rules. Only *rules* are affected by the `-r` option; default variables remain in effect (see also “Variables Used by Implicit Rules” on page 179).

- `-s`
 - `--silent`
 - `--quiet`

Silent operation; do not print the commands as they are executed. See “Command Echoing” on page 114.
- `-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` or if you set `-k` in `MAKEFLAGS` in your environment (see “Recursive Use of the `make` Tool” on page 119).
- `-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 “Instead of Executing the Commands” on page 162.
- `-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 “Recursive Use of the `make` Tool” on page 119. (In practice, you rarely need to specify this option since `make` does it for you; see “The `--print-directory` Option” on page 123.)
 - `--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 “The `--print-directory` Option” on page 123.
- `-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 “Instead of Executing the Commands” on page 162.

`--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.

11

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 a C compiler. The following documentation describes in more detail the rules of remaking target files.

- “Using Implicit Rules” on page 174
- “Catalogue of Implicit Rules” on page 175
- “Variables Used by Implicit Rules” on page 179
- “Chains of Implicit Rules” on page 181
- “Defining and Redefining Pattern Rules” on page 182
- “Defining Last-resort Default Rules” on page 188
- “Old-fashioned Suffix Rules” on page 189
- “Implicit Rule Search Algorithm” on page 191

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 “Chains of Implicit Rules” on page 181.

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 “Variables Used by Implicit Rules” on page 179.

You can define your own implicit rules by writing *pattern rules*. See “Defining and Redefining Pattern Rules” on page 182.

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 “Old-fashioned Suffix Rules” on page 189.

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 the following specification.

```
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) which 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 “Catalogue of Implicit Rules” on page 175 for a catalogue of all the predefined implicit rules.

An implicit rule applies if the required dependencies exist or can be made, and files can be made if the rule 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 “Chains of Implicit Rules” on page 181.

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 “Implicit Rule Search Algorithm” on page 191 for the details of how the search is done.

IMPORTANT! Explicit dependencies do not influence implicit rule search. For example, consider the 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 “Catalogue of Implicit Rules” on page 175).

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 “Using Empty Commands” on page 125).

Catalogue of Implicit Rules

The following is a catalogue of predefined implicit rules which are always available unless the makefile explicitly overrides or cancels them. See “Canceling Implicit Rules” on page 188 for information 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 (in the following descriptions) 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 “Old-fashioned Suffix Rules” on page 189 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 of the form, `$(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 GNU assembler. 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 the following.

```
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 is how it is 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 Ccode 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 the following:

```
.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 shown in the following example's input.

```
$(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

To make `n.dvi` from either `n.texinfo`, `n.texi`, or `n.txinfo`, use the command:

```
$(TEXI2DVI) $(TEXI2DVI_FLAGS)
```

To make `n.info` from either `n.texinfo`, `n.texi`, or `n.txinfo`, use the command in the form:

```
$(MAKEINFO) $(MAKEINFO_FLAGS)
```

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 the following.

```
$(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 “Match-anything Pattern Rules” on page 187), 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 the following.

```
$(GET) $(GFLAGS)
```

The rules for SCCS are terminal (see “Match-anything Pattern Rules” on page 187), 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 catalogue of implicit rules; for documentation on variables, see “Variables Used by Implicit Rules” on page 179.

However, the commands in built-in implicit rules actually use variables such as `COMPILE.x`, `LINK.p`, and `PREPROCESS.S`, whose values contain the commands listed in the catalogue of implicit rules.

`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 “Searching Directories for Dependencies” on page 97). 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 $@
```

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 being names of programs (like `CC`).
- Those containing 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.

The following variables are 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`.

The following are variables whose values are additional arguments for the previous

list of programs associated with variables. 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 GNU 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.

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 "Interrupting or Killing the make Tool" on page 118.

A chain can involve more than two implicit rules. For example, it is possible to make a file `f.o` from `RCS/f.o.y,v` by running `RCS`, `Yacc` and `cc`. Then both `f.o.y` and `f.o.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 `f.o` from `f.o.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 `f.o` from `f.o.c` could be handled by compiling and linking with separate chained rules, using `f.o.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.

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 non-empty 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`.

IMPORTANT! Expansion using `%` in pattern rules occurs after any variable or function expansions, which take place when the makefile is read. See "How to Use Variables" on page 127 and "Functions for Transforming Text" on page 147.

Fundamentals of 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 following form 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.

```
%.o : %.c ; command...
```

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 “Defining Last-resort Default Rules” on page 188.

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. However, 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.

Pattern Rule Examples

The following are some examples of pattern rules actually predefined in `make`.

The following shows the rule that compiles `.c` files into `.o` files:

```
%.o : %.c
      $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

This 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 “Automatic Variables” on page 184).

The following is a second built-in rule:

```
% :: RCS/%,v
    $(CO) $(COFLAGS) $<
```

This statement 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*, meaning that its dependency may not be an intermediate file (see “Match-anything Pattern Rules” on page 187). The following pattern rule has two targets:

```
%.tab.c %.tab.h: %.y
    bison -d $<
```

This tells *make* that the command `bison -dx.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 then execute.

Automatic Variables

If you are writing a pattern rule to compile a *.c* file into a *.o* file, you will need to know how to 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*, *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. For instance, you would use `$(@)` for the object file name and `$(<)` for the source file name. The following is a list of automatic variables.

`$(@)`

The file name of the target of the rule. If the target is an archive member, then `$(@)` is the name of the archive file. In a pattern rule that has multiple targets (see “Fundamentals of Pattern Rules” on page 182), `$(@)` 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 “Using *make* to Update Archive Files” on page 193. 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.

- `$(`
 The name of the first dependency. If the target got its commands from an implicit rule, this will be the first dependency added by the implicit rule (see “Using Implicit Rules” on page 174).
- `$(?`
 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 “Using make to Update Archive Files” on page 193).
- `$(^`
 The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used (see “Using make to Update Archive Files” on page 193). A target has only one dependency on each other file it depends on, no matter how many times each file is listed as a dependency. So if you list a dependency more than once for a target, the value of `$(^` 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.
- `$(*`
 The stem with which an implicit rule matches (see “How Patterns Match” on page 187). If the target is `dir/a.foo.b` and the target pattern is `a.%.b` then the stem is `dir/foo`. The stem is useful for constructing names of related files.
- In a static pattern rule, the stem is part of the file name that matched the `%` in the target pattern.
- In an explicit rule, there is no stem; so `$(*` cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see “Old-fashioned Suffix Rules” on page 189), `$(*` is set to the target name minus the suffix. For example, if the target name is `foo.c`, then `$(*` 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 previously listed, 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 “Functions for File Names” on page 151).

IMPORTANT! The `F` variants all omit the trailing slash that always appears in the output of the `dir` function.

The following is a list 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 instance.

`$(%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 “Archive Members as Targets” on page 194.

`$(<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.

We use a special stylistic convention when we discuss these automatic variables; we write “the value of `$<`”, rather than “the variable, `<`” as we would write for ordinary variables such as `objects` and `CFLAGS`. We think this convention looks more natural in this special case. Do not assume it has a deep significance; `$<` refers to the variable named `<` just as `$(CFLAGS)` refers to the variable named `CFLAGS`. You could just as well use `$(<)` in place of `$<`.

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.

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 `f.o.c,v.o` or from `RCS/SCCS/s.f.o.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 non-terminal 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 `f.o.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 `f.o.y`), the fact that its target matches is enough to prevent consideration of any non-terminal match-anything rules for the file `f.o.c`. Thus, `make` will not even consider trying to make `f.o.c` as an executable file from `f.o.c.o`, `f.o.c.c`, `f.o.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 `f.o.p` match a specific target pattern and thereby prevent time from being wasted looking for `f.o.p.o` or `f.o.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 “Old-fashioned Suffix Rules” on page 189).

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
```

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 “Match-anything Pattern Rules” on page 187). 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 the following.

```
% ::
    touch $@
```

This causes 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 like: `.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 “Using Empty Commands” on page 125).

You can use a last-resort rule to override part of another makefile (see “Overriding Part of Another Makefile” on page 92).

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 `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`. The following 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, use the following rule.

```
.c.o: foo.h
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

This rule tells how to make the file, `.c.o`, from the dependency file, `foo.h`, and is not at all like the following pattern rule.

```
%.o: %.c foo.h
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

This rule 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 “Canceling Implicit Rules” on page 188). 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, use the following.

```
.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.

Implicit Rule Search Algorithm

The following 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)`, run the following algorithm 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 non-terminal 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 non-empty 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.
4. 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.

6. 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 database 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 “Automatic Variables” on page 184.

12

Using `make` to Update Archive Files

Archive files are files containing named subfiles called *members*; they are maintained with the binary utility, `ar`, and their main use is as subroutine libraries for linking.

The following documentation discusses `make`'s updating of your archive files.

- “Archive Members as Targets” (below)
- “Implicit Rule for Archive Member Targets” on page 194
- “Updating Archive Symbol Directories” on page 195
- “Dangers When Using Archives” on page 195
- “Suffix Rules for Archive Files” on page 196

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 instance, this rule says to create a member, `hack.o`, in archive, `foolib`, by copying the file, `hack.o` as in the following.

```
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.

IMPORTANT! The `c` flag to `ar` is required if the archive file does not already exist.

To specify several members in the same archive, write all the member names together between the parentheses, as in the following example.

```
foolib(hack.o kludge.o)
```

The previous statement is equivalent to the following statement.

```
foolib(hack.o) foolib(kludge.o)
```

You can also use shell-style wildcards in an archive member reference. See “Using Wildcard Characters in File Names” on page 95. 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).
```

Implicit Rule for Archive Member Targets

Recall that a target that looks like $a(m)$ stands for the member, *min*, 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 parentheses 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
```

In the previous example, `make` has envisioned the `bar.o` as an intermediate file (see also “Chains of Implicit Rules” on page 181). Implicit rules such as this one are written using the automatic variable, `$(` (see “Automatic Variables” on page 184).

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 the command, `ar r foo.a dir/file.o`, having 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.

Updating Archive Symbol Directories

An archive file that is used as a library usually contains a special member named `__.SYMDEF` which 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. Use the following example, for instance.

```
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 (see “Implicit Rule for Archive Member Targets” on page 194 for more information).

This is not necessary when using the GNU `ar` program which automatically updates the `__.SYMDEF` member.

Dangers When Using Archives

It is important to be careful when using parallel execution (the `-j` switch; see “Parallel Execution” on page 116) 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`.

Suffix Rules for Archive Files

You can write a special kind of suffix rule for with archive files. See “Old-fashioned Suffix Rules” on page 189 for a full explanation of suffix rules.

Archive suffix rules are obsolete in `make` because pattern rules for archives are a more general mechanism (see “Implicit Rule for Archive Member Targets” on page 194 for more information). But they are retained for compatibility with other `makes`.

To write a suffix rule for archives, you write a suffix rule using the target suffix, `.a` (the usual suffix for archive files). For instance, the following example shows the 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 following 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 “Old-fashioned Suffix Rules” on page 189). Thus a double-suffix rule, `.x.a`, produces two pattern rules: `(%.o): %.x` and `%.a: %.x`.

13

Summary of the Features for the GNU `make` utility

The following summary describes the features of GNU `make` compared to other versions of `make`. For comparison purposes, the features of `make` in 4.2 BSD systems act as a baseline. If you are concerned with writing portable makefiles, consider the following features of `make`, using them with caution; see also “GNU `make`’s Incompatibilities and Missing Features” on page 201. Many features come from the System V version of `make`.

- The `VPATH` variable and its special meaning. 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). See “Searching Directories for Dependencies” on page 97.
- Included makefiles. See “Including Other Makefiles” on page 89. Allowing multiple files to be included with a single directive is a GNU extension.

- Variables are read from and communicated, using the environment. See “Variables from the Environment” on page 138.
- Options passed through the variable `MAKEFLAGS` to recursive invocations of `make`. See “Communicating Options to a Sub-make Utility” on page 122.
- The automatic variable, `$$`, is set to the member name in an archive reference. See “Automatic Variables” on page 184.
- The automatic variables, `$(@)`, `$(*)`, `$(<)`, `$(%)`, and `$(?)`, have corresponding forms like `$(@F)` and `$(@D)`. We have generalized this to `$$^` as an obvious extension. See “Automatic Variables” on page 184.
- Substitution variable references. See “Basics of Variable References” on page 128.
- The command-line options, `-b` and `-m`, are accepted and ignored. In System V `make`, these options actually do something.
- Execution of recursive commands to run `make`, using the variable `MAKE` even if `-n`, `-q` or `-t` is specified. See “Recursive Use of the make Tool” on page 119.
- Support for suffix `.a` in suffix rules. See “Suffix Rules for Archive Files” on page 196. This feature is obsolete in GNU `make` because the general feature of rule chaining (see “Chains of Implicit Rules” on page 181) allows one pattern rule for installing members in an archive (see “Implicit Rule for Archive Member Targets” on page 194) 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`.

- Pattern rules using `%`. This has been implemented in several versions of `make`. See “Defining and Redefining Pattern Rules” on page 182.
- 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”). See “Chains of Implicit Rules” on page 181.
- The automatic variable, `$$^`, containing a list of all dependencies of the current target. See “Automatic Variables” on page 184. The automatic variable, `$$+`, is a simple extension of `$$^`.
- The *what if* flag (`-w` in GNU `make`) was invented by Andrew Hume in `mk`. See “Instead of Executing the Commands” on page 162.
- 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 “Command Execution” on page 114.
- Modified variable references using pattern substitution come from SunOS 4. See

“Basics of Variable References” on page 128. 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 “Instead of Executing the Commands” on page 162) 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 “Appending More Text to Variables” on page 135.
- The syntax `archive(mem1 mem2 ...)` to list multiple members in a single archive file comes from SunOS 4 `make`. See “Implicit Rule for Archive Member Targets” on page 194.
- 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 “The Two Flavors of Variables” on page 129.
- Pass command-line variable assignments automatically through the variable, `MAKE`, to recursive `make` invocations. See “Recursive Use of the `make` Tool” on page 119.
- Use the `-C` or `--directory` command option to change directory. See “Summary of `make` Options” on page 167.
- Make verbatim variable definitions with `define`. See “Defining Variables Verbatim” on page 137.
- 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 “Phony Targets” on page 101.
- Manipulate text by calling functions. See “Functions for Transforming Text” on page 147.
- Use the `-o` or `--old-file` option to pretend a file’s modification-time is old. See “Avoiding Recompilation of Some Files” on page 164.
- Conditional execution 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 “Conditional Parts of Makefiles” on page 141.
- Specify a search path for included makefiles. See “Including Other Makefiles”

- on page 89.
- Specify extra makefiles to read with an environment variable. See “The `MAKEFILES` Variable” on page 90.
- 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 “Directory Search for Link Libraries” on page 101.
- Allow suffixes for suffix rules (see “Old-fashioned Suffix Rules” on page 189) 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 “Recursive Use of the `make` Tool” on page 119.
- Specify static pattern rules. See “Static Pattern Rules” on page 107.
- Provide selective `vpath` search. See “Searching Directories for Dependencies” on page 97.
- Provide computed variable references. See “Basics of Variable References” on page 128.
- Update makefiles. See “How Makefiles are Remade” on page 91. 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 “Catalogue of Implicit Rules” on page 175.
- The built-in variable, `MAKE_VERSION`, gives the version number of `make`.

14

GNU `make`'s Incompatibilities and Missing Features

The following documentation describes some incompatibilities and missing features in GNU `make`. See also “Problems and Bugs with make Tools” on page 203. 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*) that specifies `make` does not require any of these features.

- A target of the form, `file((entry))`, standing for a member of archive file, `file`. The member is chosen, not by name, but by being an object file that defines the linker symbol, `entry`. This feature was not put into GNU `make` because of the non-modularity of putting knowledge into `make` of the internal format of archive file symbol tables. See “Updating Archive Symbol Directories” on page 195.
- Suffixes (used in suffix rules) ending with the `~` character have a special meaning to System V `make`; they refer to the SCCS (Source Code Control System) 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 “Old-fashioned Suffix Rules” on page 189.

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 “Chains of Implicit Rules” on page 181.

- In System V `make`, the string, `$$@`, has the meaning that, in the dependencies of a rule with multiple targets, it stands for the particular target that is being processed. This is not defined in GNU `make` because `$$` should always stand for an ordinary `$`. It is possible to get this functionality through the use of static pattern rules (see “Static Pattern Rules” on page 107).

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 “Searching Directories for Dependencies” on page 97 and “VPATH: Search Path for All Dependencies” on page 98) 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 `makes`, the automatic variable, `$*`, appearing in the dependencies of a rule, has the feature of expanding to the full name of the target of that rule, inconsistent with the normal definition of `$*`.
- In some `makes`, implicit rule search is apparently done for all targets, not just those without commands (see “Using Implicit Rules” on page 174). This means you can use `foo.o: cc -c foo.c`, and `make` will intuit that `foo.o` depends on `foo.c`.

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.
- 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 “Using Empty Commands” on page 125). For example, `.c.a:` will override the built-in `.c.a` suffix rule.
- Some versions of `make` invoke the shell with the `-e` flag, except under `-k` (see “Testing the Compilation of a Program” on page 165). The `-e` flag tells the shell to exit as soon as any program it runs returns a nonzero status; it is cleaner to write each shell command line to stand on its own without requiring this special treatment.

Problems and Bugs with `make` Tools

If you have problems with GNU `make` or think you've found a bug, please report it to the developers. Once you've got a precise problem, please send email:

`bug-make@gnu-org`

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`, generated by the configuration process.

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 the makefile and the exact results `make` gave you. Also explain what you expected to occur; this will help to determine whether the problem was really in the documentation.

15

Makefile Conventions

The following discusses conventions for writing Makefiles for GNU programs.

- “General Conventions for Makefiles” (below)
- “Utilities in Makefiles” on page 207
- “Standard Targets for Users” on page 207
- “Variables for Specifying Commands” on page 211
- “Variables for Installation Directories” on page 212
- “Install Command Categories” on page 216

General Conventions for Makefiles

Every Makefile should contain the following line to avoid trouble on systems where

the `SHELL` variable might be inherited from the environment.

```
SHELL=/bin/sh
```

Different `make` programs have incompatible suffix lists and implicit rules. So it is a good idea to set the suffix list explicitly using only the suffixes you need in the particular Makefile, using something like the following example shows.

```
.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, make sure to use `./` 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 `./` (signifying the *build* directory) and `$(srcdir)/` (signifying the *source* directory) is important when using the `--srcdir` option to `configure`. A rule of the following form will fail when the build directory is not the source directory, because `foo.man` and `sedscript` are in the source directory.

```
foo.1 : foo.man sedscript
sed -e sedscript foo.man > foo.1
```

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 the following should instead be re-written in order to allow `VPATH` to work correctly.

```
foo.o : bar.c
$(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

The makefile should be written like the following example input shows.

```
foo.o : bar.c
$(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

When the target has multiple dependencies, using an explicit `\$(srcdir)` is the easiest way to make the rule work well. For instance, the previous target for `foo.1` is best written as the following example input shows.

```
foo.1 : foo.man sedscript
      sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

GNU distributions usually contain some files which are not source files; for example, info files, and the output from `autoconf`, `automake`, `Bison` or `Flex`. Since these files normally appear in the source directory, they should always appear in the source directory, not in the build directory. So Makefile rules to update them should put the updated files in the source directory.

However, if a file does not appear in the distribution, then the Makefile should not put it in the source directory, because building a program in ordinary circumstances

should not modify the source directory in any way.

Try to make the build and installation targets, at least (and all their subtargets) work correctly with a parallel `make`.

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 the following:

```
cat cmp cp diff echo egrep expr false grep install-info
ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true
```

The compression program, `gzip`, can be used in the `dist` rule.

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 using `make` variables so that the user can substitute alternatives.

The following are some of the programs.

```
ar bison cc flex install ld ldconfig lex
make makeinfo ranlib texi2dvi yacc
```

Use the following `make` variables:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

When you use `ranlib` or `ldconfig`, you should make sure nothing bad happens if the system does not have the program in question. Arrange to ignore an error from that command, and print a message before the command to tell the user that failure of this command does not mean a problem. The `AC_PROG_RANLIB` `autoconf` macro can help with this problem.

If you use symbolic links, you should implement a fallback for systems that don't have symbolic links.

The following utilities also use the `make` variables.

```
chgrp chmod chown mknod
```

It is acceptable to use other utilities in Makefile portions (or scripts) intended only for particular systems where you know those utilities to exist.

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 sub-directories that are needed. One way to do this is by means of an `installdirs` target.

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 "Variables for Specifying Commands" on page 211), 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. The following 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 non-installed 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, `maintainer-clean` should start with the following two commands.

```
@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 in the Texinfo package called `mkinstalldirs` that is convenient for this functionality. You can use a rule like the following example shows.

```

# 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.

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 the GNU linker.

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 the following example input.

```

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.

`CFLAGS` should be used in every invocation of the C compiler, both those which do compilation and those which do linking.

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 the following example input shows.

```
$(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.

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 in the following documentation. 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 in the following discussions for installing directories. 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 `.` (With `autoconf`, use the `@prefix@` variable.)

`exec_prefix`

A prefix used in constructing the default values of some of the variables listed in the following discussions for installing directories. 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. (With `autoconf`, use the `@exec_prefix@` variable.)

Executable programs are installed in one of the following directories.

`bindir`

The directory for installing executable programs users run. This should normally be `/usr/local/bin` but write it as `\$(exec_prefix)/bin`. (With `autoconf`, use the `@bindir@` variable.)

`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`. With `autoconf`, use the `@sbindir@` variable.)

`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`. With `autoconf`, use the `@libexecdir@` variable.)

Data files used by `make` during its execution are divided into categories in the following 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 difficult.

Therefore, the following variables are what 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)` in the following discussions for them. (With `autoconf`, use the `@datadir@` variable.)

`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)`. (With `autoconf`, use the `@sysconfdir@` variable.)

`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`. (With `autoconf`, use the `@sharedstatedir@` variable.)

`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`. (With `autoconf`, use the `@localstatedir@` variable.)

`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`. (With `autoconf`, use the `@libdir@` variable.)

`lispdir`

The directory for installing any Emacs Lisp files in this package. By default, it should be `/usr/local/share/emacs/site-lisp`, but it should be written as `\$(prefix)/share/emacs/site-lisp`. If you are using `autoconf`, write the default as `@lispdir@`. In order to make `@lispdir@` work, you need the following lines in your `configure.in` file:

```
lispdir='${datadir}/emacs/site-lisp'  
AC_SUBST(lispdir)
```

`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`. (With `autoconf`, use the `@infodir@` variable.)

`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`. (With `autoconf`, use the `@includedir@` variable.)

`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. (With `autoconf`, use the `@oldincludedir@` variable.)

`man` pages are installed in one of the following directories.

`mandir`

The directory for installing the man pages (if any) for this package. It should include the suffix for the proper section of the documentation—usually 1 for a utility. It will normally be `/usr/local/man/man1` but you should write it as `\$(prefix)/man/man1`. (With `autoconf`, use the `@mandir@` variable.)

`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 documentation.

WARNING! Don't make the primary documentation for any GNU software be a man page. Using Emacs, write documentation in Texinfo instead. man pages are just for the sake of people running GNU software, and only a secondary application.

`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 documentation.

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. (With `autoconf`, use the `srcdir = @srcdir@` variable.) Use the following example's input, for instance.

```
# 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 previously discussed. 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.

Install Command Categories

When writing the `install` target, you must classify all the commands into three categories: normal ones, pre-installation commands and post-installation commands.

Normal commands move files into their proper places, and set their modes. They may not alter any files except the ones that come entirely from the package to which they belong.

Pre-installation and post-installation commands may alter other files; in particular, they can edit global configuration files or data bases.

Pre-installation commands are typically executed before the normal commands, and post-installation commands are typically run after the normal commands.

The most common use for a post-installation command is to run `install-info`. This cannot be done with a normal command, since it alters a file (the Info directory) which does not come entirely and solely from the package being installed. It is a post-installation command because it needs to be done after the normal command which installs the package's Info files.

Most programs don't need any pre-installation commands, but the feature is available just in case it is needed.

To classify the commands in the `install` rule into these three categories, insert *category lines* among them. A category line specifies the category for the commands that follow.

A category line consists of a tab and a reference to a special `make` variable, plus an optional comment at the end. There are three variables you can use, one for each category; the variable name specifies the category. Category lines are no-ops in ordinary execution because these three `make` variables are normally undefined (and you should not define them in the makefile).

The following documentation discusses the three possible category lines.

```
$(PRE_INSTALL)
    # Pre-install commands follow.
$(POST_INSTALL)
    # Post-install commands follow.
$(NORMAL_INSTALL)
    # Normal commands follow.
```

If you don't use a category line at the beginning of the `install` rule, all the commands are classified as normal until the first category line.

If you don't use any category lines, all the commands are classified as normal.

The following category lines are for `uninstall`.

```
$(PRE_UNINSTALL)
    # Pre-uninstall commands follow.
$(POST_UNINSTALL)
    # Post-uninstall commands follow.
$(NORMAL_UNINSTALL)
    # Normal commands follow.
```

Typically, a pre-uninstall command would be used for deleting entries from the Info directory.

If the `install` or `uninstall` target has any dependencies which act as subroutines of installation, then you should start each dependency's commands with a category line, and start the main target's commands with a category line also. This way, you can ensure that each command is placed in the right category regardless of which of the

dependencies that actually run.

Pre-installation and post-installation commands should not run any programs except for the following utilities.

```
basename bash cat chgrp chmod chown cmp cp dd diff echo
egrep expand expr false fgrep find getopt grep gunzip gzip
hostname install install-info kill ldconfig ln ls md5sum
mkdir mkfifo mknod mv printenv pwd rm rmdir sed sort tee
test touch true uname xargs yes
```

The reason for distinguishing the commands in this way is for the sake of making binary packages. Typically a binary package contains all the executables and other files that need to be installed, and has its own method of installing them; so, it does not need to run the normal installation commands. Installing the binary package *does* need to execute the pre-installation and post-installation commands.

Programs to build binary packages work by extracting the pre-installation and post-installation commands. The following example's input shows one way of extracting the pre-installation commands.

```
make -n install -o all \
    PRE_INSTALL=pre-install \
    POST_INSTALL=post-install \
    NORMAL_INSTALL=normal-install \
    gawk -f pre-install.awk
```

The `pre-install.awk` file could contain the following pre-installation commands.

```
$0 ~ /Ä\t[ \t]*(normal_install|post_install)[ \t]*$/ {on = 0}
on {print $0}
$0 ~ /Ä\t[ \t]*pre_install[ \t]*$/ {on = 1}
```

The resulting file of pre-installation commands is executed as a shell script as part of installing the binary package.

16

GNU `make` Quick Reference

The following documentation describes the directives, text manipulation functions, special variables that `make` understands and recognizes, and error messages that `make` generates and what they mean.

- “Directives that `make` Uses” on page 220
- “Text Manipulation Functions” on page 221
- “Automatic Variables that `make` Uses” on page 222
- “Variables that `make` Uses” on page 223
- “Error Messages that `make` Generates” on page 224

See also “Special Built-in Target Names” on page 104, “Catalogue of Implicit Rules” on page 175, and “Summary of `make` Options” on page 167 for other discussions.

Directives that make Uses

```
define variable
endif
```

Define a multi-line, recursively-expanded variable. See “Defining Canned Command Sequences” on page 124.

```
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 “Conditional Parts of Makefiles” on page 141.

```
include file
```

Include another makefile. See “Including Other Makefiles” on page 89.

```
override variable= value
override variable:= value
override variable+= value
override define variable
endif
```

Define a variable, overriding any previous definition, even one from the command line. See “The override Directive” on page 137.

```
export
```

Tell make to export all variables to child processes by default. See “Communicating Variables to a Sub-make Utility” on page 120.

```
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 “Communicating Variables to a Sub-make Utility” on page 120.

```
vpath pattern path
```

Specify a search path for files matching a % pattern. See “The vpath Directive” on page 98.

`vpath pattern`

Remove all search paths previously specified for *pattern*.

`vpath`

Remove all search paths previously specified in any `vpath` directive.

Text Manipulation Functions

The following is a summary of the text manipulation functions (see “Functions for Transforming Text” on page 147):

`$(subst from, to, text)`

Replace *from* with *to* in *text*. See “Functions for String Substitution and Analysis” on page 148.

`$(patsubst pattern, replacement, text)`

Replace words matching *pattern* with *replacement* in *text*. See “Functions for String Substitution and Analysis” on page 148.

`$(strip string)`

Remove excess whitespace characters from *string*. See “Functions for String Substitution and Analysis” on page 148.

`$(findstring find, text)`

Locate *find* in *text*. See “Functions for String Substitution and Analysis” on page 148.

`$(filter pattern... , text)`

Select words in *text* that match one of the *pattern* words. See “Functions for String Substitution and Analysis” on page 148.

`$(filter-out pattern... , text)`

Select words in *text* that do not match any of the *pattern* words. See “Functions for String Substitution and Analysis” on page 148.

`$(sort list)`

Sort the words in *list* lexicographically, removing duplicates. See “Functions for String Substitution and Analysis” on page 148.

`$(dir names...)`

Extract the directory part of each file name. See “Functions for File Names” on page 151.

`$(notdir names...)`

Extract the non-directory part of each file name. See “Functions for File Names” on page 151.

`$(suffix names...)`

Extract the suffix (the last `.` and the characters that follow it) of each file name. See “Functions for File Names” on page 151.

`$(basename names...)`

Extract the base name (name without suffix) of each file name. See “Functions for File Names” on page 151.

`$(addsuffix suffix, names...)`

Append *suffix* to each word in *names*. See “Functions for File Names” on page 151.

`$(addprefix prefix, names...)`

Prepend *prefix* to each word in *names*. See “Functions for File Names” on page 151.

`$(join list1, list2)`

Join two parallel lists of words. See “Functions for File Names” on page 151.

`$(word n, text)`

Extract the *n*th word (one-origin) of *text*. See “Functions for File Names” on page 151.

`$(words text)`

Count the number of words in *text*. See “Functions for File Names” on page 151.

`$(firstword names...)`

Extract the first word of *names*. See “Functions for File Names” on page 151.

`$(wildcard pattern...)`

Find file names matching a shell file name, *pattern* (not a % pattern). See “The wildcard Function” on page 96.

`$(shell command)`

Execute a shell command and return its output. See “The shell Function” on page 156.

`$(origin variable)`

Return a string describing how the make variable, *variable*, was defined. See “The origin Function” on page 155.

`$(foreach var, words, text)`

Evaluate *text* with *var* bound to each word in *words*, and concatenate the results. See “The foreach Function” on page 153.

Automatic Variables that make Uses

The following is a summary of the automatic variables. See “Automatic Variables” on page 184 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 “Using `make` to Update Archive Files” on page 193).
- `$^`
`$+`
The names of all the dependencies with spaces between them. For dependencies which are archive members, only the member named is used (see “Using `make` to Update Archive Files” on page 193). The value of `$^` omits duplicate dependencies while `$+` retains them and preserves their order.
- `$*`
The stem with which an implicit rule matches (see “How Patterns Match” on page 187).
- `$(@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)`
`$(?F)`
The directory part and the file-within-directory part of `$?`.

Variables that `make` Uses

The following variables are used specially by GNU `make`.

`MAKEFILES`

Makefiles to be read on every invocation of `make`. See “The `MAKEFILES` Variable” on page 90.

VPATH

Directory search path for files not found in the current directory.

See “VPATH: Search Path for All Dependencies” on page 98.

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 “Command Execution” on page 114.

MAKE

The name with which `make` was invoked. Using this variable in commands has special meaning. See “How the MAKE Variable Works” on page 119.

MAKESHELL

On MS-DOS only, the name of the command interpreter that is to be used by `make`. This value takes precedence over the value of `SHELL`. See “Command Execution” on page 114.

MAKELEVEL

The number of levels of recursion (sub-`makes`). See “Communicating Variables to a Sub-make Utility” on page 120.

MAKEFLAGS

The flags given to `make`. You can set this in the environment or a makefile to set flags. See “Communicating Variables to a Sub-make Utility” on page 120.

MAKECMDGOALS

The targets given to `make` on the command line. Setting this variable has no effect on the operation of `make`. See “Arguments to Specify the Goals” on page 160.

CURDIR

Set to the pathname of the current working directory (after all `-C` options are processed, if any). Setting this variable has no effect on the operation of `make`. See “Recursive Use of the make Tool” on page 119.

SUFFIXES

The default list of suffixes before `make` reads any makefiles.

Error Messages that `make` Generates

The following documentation shows the most common errors you might see generated by `make`, and discusses some information about what they mean and how to fix them.

Sometimes, `make` errors are not fatal, especially in the presence of a dash (`-`) prefix on a command script line, or the `-k` command line option. Errors that are fatal are prefixed with the string, `***`, and error messages are all either prefixed with the name of the program (usually `make`), or, if the error is found in a makefile, the name of the file and linenummer containing the problem.

[foo] Error NN

[foo] signal description

These errors do not really make errors at all. They mean that a program that `make` invoked as part of a command script returned a non-0 error code (**Error NN**), which `make` interprets as failure, or it exited in some other abnormal fashion (with a signal of some type).

If no ******* is attached to the message, then the subprocess failed but the rule in the makefile was prefixed with the special character dash (-), so `make` ignored the error.

missing separator. Stop.

This is `make`'s generic "Huh?" error message. It means that `make` was completely unsuccessful at parsing this line of your makefile. It basically means a syntax error.

One of the most common reasons for this message is that the command scripts begin with spaces instead of a TAB character (as is the case with many scripts viewed in MS-Windows editors). Every line in the command script must begin with a TAB character. Eight spaces do not count.

commands commence before first target. Stop.

missing rule before commands. Stop.

This means the first thing in the makefile seems to be part of a command script: it begins with a TAB character and doesn't appear to be a legal `make` command (such as a variable assignment). Command scripts must always be associated with a target.

The second form is generated if the line has a semicolon as the first non-whitespace character; `make` interprets this to mean you left out the "target: dependency" section of a rule.

No rule to make target 'xxx'.

No rule to make target 'xxx', needed by 'yyy'.

This means that `make` decided it needed to build a target, and any instructions in the makefile weren't found by `make` for execution of that process, either explicitly or implicitly (including in the default rules database).

If you want that file to be built, you will need to add a rule to your makefile describing how that target can be built. Other possible sources of this problem are typing errors in the makefile (if, for instance, a filename is wrong) or using a corrupted source tree (if a specific file is not intended to be built, but rather that the file is only a dependency).

No targets specified and no makefile found. Stop.

No targets. Stop.

The former means that you didn't provide any targets to be built on the command line, and `make` couldn't find any makefiles to read. The latter means that some makefile was found, and it didn't contain any default target and no target was given on the command line. `make` has nothing to do in these situations.

Makefile 'xxx' was not found.**Included makefile 'xxx' was not found.**

A makefile specified on the command line (the first form) or included (the second form) was not found.

warning: overriding commands for target 'xxx'**warning: ignoring old commands for target 'xxx'**

`make` allows commands to be specified only once per target (except for double-colon rules). If you give commands for a target which already has been defined to have commands, this warning is issued and the second set of commands will overwrite the first set.

Circular xxx <- yyy dependency dropped.

This means that `make` detected a loop in the dependency graph; after tracing the dependency, `yyy`, of target, `xxx`, and its dependencies, one of them depended on `xxx` again.

Recursive variable 'xxx' references itself (eventually). Stop.

This means you've defined a normal (recursive) `make` variable, `xxx`, that, when it's expanded, will refer to itself (`xxx`). This is not allowed; either use simply-expanded variables (such as `=`) or use the append operator (`+=`).

Unterminated variable reference. Stop.

This means you forgot to provide the proper closing parenthesis or brace in your variable or function reference.

insufficient arguments to function 'xxx'. Stop.

This means you haven't provided the requisite number of arguments for this function. See the documentation of the specific function for a description of its arguments.

missing target pattern. Stop.**multiple target patterns. Stop.****target pattern contains no '%'. Stop.**

These are generated for malformed static pattern rules. The first means there's no pattern in the target section of the rule, the second means there are multiple patterns in the target section, and the third means the target doesn't contain a pattern character (`%`).

17

Complex Makefile Example

The following is the makefile for the GNU `tar` program, 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; not shown in this documentation.

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 ioctl's).
# -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/id' \  
-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.dirIndex
```

Using `diff` & `patch`

Copyright © 1988-2000 Free Software Foundation

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

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the “GNU General Public License” 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 documentation into another language, under the above conditions for modified versions.

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” by Eugene W. Myers, in *Algorithmica*; Vol. 1, No. 2, 1986; pp. 251–266; and in “A File Comparison Program” by Webb Miller and Eugene W. Myers, in *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” by E. Ukkonen, in *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 the documentation are adapted from a material written by Larry Wall, with his permission.

Copyright © 1992-2000 Red Hat.

GNUPro[®], the GNUPro[®] logo, and the Red Hat[®] logo are trademarks of Red Hat.

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

All rights reserved.

1

Overview of `diff` & `patch`, the Compare & Merge Tools

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 initially were identical copies that were then 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*.

The following documentation discusses using the commands and other related commands.

- “What Comparison Means” on page 237
- “diff Output Formats” on page 243
- “Comparing Directories” on page 261
- “Making diff Output Prettier” on page 263

- “`diff` Performance Tradeoffs” on page 265
- “Comparing Three Files” on page 267
- “Merging from a Common Ancestor” on page 271
- “`sdiff` Interactive Merging” on page 277
- “Merging with the `patch` Utility” on page 281
- “Tips for Making Distributions with Patches” on page 287
- “Invoking the `cmp` Utility” on page 289
- “Invoking the `diff` Utility” on page 291
- “Invoking the `diff3` Utility” on page 299
- “Invoking the `patch` Utility” on page 303
- “Invoking the `sdiff` Utility” on page 311
- “Incomplete Lines” on page 315
- “Future Projects for `diff` and `patch` Utilities” on page 317

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, **Meta-x** `compare-windows`. See “Comparing Files” in *The GNU Emacs Manual** for more information, particularly 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 documentation first concentrates on making `diffs`, and later shows how to use `diffs` to update files.

* *The GNU Emacs Manual* is published by the Free Software Foundation (ISBN 1-882114-03-5).

2

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 another file. `diff` compares two files line by line, finding groups of lines that differ, and then reporting each group of differing lines. It can report the differing lines in several formats, each of which have different purposes.

See the following documentation for more information.

- “Hunks” on page 238
- “Suppressing Differences in Blank and Tab Spacing” on page 239
- “Suppressing Differences in Blank Lines” on page 239
- “Suppressing Case Differences” on page 240

- “Suppressing Lines Matching a Regular Expression” on page 240
- “Summarizing Which Files Differ” on page 240
- “Binary Files and Forcing Text Comparisons” on page 241

`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. 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.

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 the following output:

```
1,2d0
< a
< b
3a2,3
> b
> a
```

But if `diff` notices the common line `b` instead, it produces the following 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 “diff Performance Tradeoffs” on page 265).

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 additional 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§
```

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 instance, consider a file containing only the following lines.

```
1. A point is that which has no part.

2. A line is breadthless length.
-- Euclid, The Elements, I
```

The last example is considered identical to another file containing only the following lines.

1. A point is that which has no part.
2. A line is breadthless length.

```
-- Euclid, The Elements, I
```

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.

Suppressing Lines Matching a Regular Expression

To ignore insertions and deletions of lines that match a regular expression, use the `-I regex` or `--ignore-matching-lines=regex` option.

You should escape regular expressions that contain shell meta-characters 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 non-ignorable 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.

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 “Invoking the `cmp` Utility” on page 289).

Unlike `diff`, `cmp` cannot compare directories; it can only compare two files.

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 “Summarizing Which Files Differ” on page 240), 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 (but not how) they differ by using the `--brief` option.

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 “Invoking the `cmp` Utility” on page 289 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.

3

`diff` Output Formats

`diff` has several mutually exclusive options for output format. The following documentation discusses the output and formats.

- “Two Sample Input Files” on page 244
- “Showing Differences Without Context” on page 244
- “Showing Differences in Their Context” on page 246
- “Showing Differences Side by Side” on page 251
- “Controlling Side by Side Format” on page 252
- “Merging Files with If-then-else” on page 255

Two Sample Input Files

The following are two sample files that we will use in numerous examples to illustrate the output of `diff` and how various options can change it. The following lines are from the ‘`lao`’ file .

```
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.
```

The following is the ‘`tzu`’ file.

```
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 the ‘`lao`’ file, while the second hunk contains the fourth line of ‘`lao`’ opposing the second and third lines of the ‘`tzu`’ file; the last hunk contains just the last three lines of the ‘`tzu`’ file.

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 “Context Format” on page 246) and the unified format (see “Unified Format” on page 248) are superior. Normal format is the default for compatibility with older versions of `diff` and the Posix standard.

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 the following excerpt:

```
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 the following.

lar

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.

fmt

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.

rdl

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.

An Example of Normal Format

The following is the output of the ‘diff lao tzu’ command (see “Two Sample Input Files” on page 244 for the complete contents of the two files).

Notice that the following example 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!
```

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 “Applying Imperfect Patches” on page 282 for more information on using `patch` to apply imperfect diffs.

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.

Detailed Description of Context Format

The context output format starts with a two-line header, which looks like the following lines.

```
*** 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 “Showing Alternate File Names” on page 250. Next come one or more hunks of differences; each hunk shows one area where the files differ. Context format hunks look like the following lines.

```
*****
*** 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.

An Example of Context Format

Here is the output of ‘diff -c lao tzu’ (see “Two Sample Input Files” on page 244 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!
```

An Example of Context Format With Less Context

The following example shows the output of `diff --context=1 lao tzu` (see “Two Sample Input Files” on page 244 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!
```

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.

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 “Showing Alternate File Names” on page 250. Next come one or more hunks of differences; each hunk shows one area where the files differ. Unified format hunks look like the following

```
@@ 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.

An Example of Unified Format

Here is the output of the command, ‘diff -u lao tzu’ (see “Two Sample Input Files” on page 244 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!
```

Showing Sections In Which There Are Differences

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.

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:

C, C++, Prolog

```
'^[A-Za-z_]'
```

Lisp

```
'^('
```

Texinfo

```
'^@\\(chapter\\|appendix\\|unnumbered\\|chapeheading\\)'
```

This option does not automatically select an output format; in order to use it, you must select the context format (see “Context Format” on page 246) or unified format (see “Unified Format” on page 248). 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.

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 “Context Format” on page 246), 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 “Showing Lines that Match Regular Expressions” on page 249).

GNU `diff` provides them for the sake of convenience.

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 “Paginating diff Output” on page 264). The following are the first two lines of the output from `diff -C2 -Loriginal -Lmodified lao tzu`:

```
*** original
--- modified
```

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 “Suppressing Differences in Blank and Tab Spacing” on page 239).

‘|’

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 “Incomplete Lines” on page 315. 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 “`sdiff` Interactive Merging” on page 277 for more information on merging files.

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.

An Example of Side by Side Format

The following is the output of the command ‘`diff -y -W 72 lao tzu`’ (see “Two Sample Input Files” on page 244 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!
```

Making Edit Scripts

Several output modes produce command scripts for editing *from-file* to produce *to-file*.

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

“Showing Differences Without Context” on page 244), 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’.

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 the following:

```
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:

‘la’

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.

‘rc’

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.

‘rd’

Delete the lines in range *r* from the first file. For example, ‘5,7d’ means delete lines 5–7 of file 1.

Example ed Script

The following is the output of `diff -e lao tzu` (see “Two Sample Input Files” on page 244 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
```

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.

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 “Forward ed Scripts” on page 254), 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. The following is the output of `diff -n lao tzu` (see “Two Sample Input Files” on page 244 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!

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 the following declaration.

```
do {
#ifdef 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.

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 previous 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
```

What follows is another example of output for 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 “Line Formats” on page 258).

‘%>’

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*’

Stands for the character with octal code *o*, where *o* is a string of 1, 2, or 3 octal digits. For example, ‘%c’ \0’ stands for a null character.

‘%*Fn*’

Stands for *n*’s value formatted with *F* where *F* is a `printf` conversion specification and *n* is one of the following letters.

‘*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.

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 simultaneously sets all three of the previous options.

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 trail-ing newline (if any). This format ignores whether the line is incomplete; see “Incomplete Lines” on page 315.

`'%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'`

Stands for `c`, where `c` is a single character. `c` may not be a backslash or an apostrophe. For example, `'%c' ':'` stands for a colon.

`'%c' \ o'`

Stands for the character with octal code `o` where `o` is a string of 1, 2, or 3 octal digits. For example, `'%c' \0'` stands for a null character.

`'Fn'`

Stands for the line number formatted with `F` where `F` is a `printf` conversion specification. For example, `%.5dn` prints the line number using the `printf` format, `"%.5d"`. See “Line Group Formats” on page 255 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=1?:,%dl)d%dE
%<' \
  --new-group-format='%dea%dF%(F=L?:,%dL)
%>' \
  --changed-group-format='%df%(f=1?:,%dl)c%dF%(F=L?:,%dL)
%<---
%>' \
  --unchanged-group-format=' ' \
old new
```

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 any-way 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 “patch Options” on page 306.

An Example of If-then-else Format

The following is the output of ‘`diff -DTWO lao tzu`’ (see “Two Sample Input Files” on page 244 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;
#ifdef 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 */
```

4

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` options 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 “Tips for Making Distributions with Patches” on page 287 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, using 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 subsequent files in the topmost directory level.

5

Making `diff` Output Prettier

`diff` provides several ways to adjust the appearance of its output. These adjustments can be applied to any output format. For more information, see “Preserving Tabstop Alignment” on page 263 and “Paginating `diff` Output” on page 264.

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`, use the `-l` or `--ignore-white-space` options (see “Applying Patches in Other Directories” on page 304 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` options.

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. The following is what the page header might look like for `diff -lc lao tzu`:

```
Mar 11 13:37 1991 diff -lc lao tzu Page 1
```

6

`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` options do 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, use the `-H` or `--speed-large-files` options 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* of the prefix and the first *lines* of the suffix. This gives `diff` further opportunities to find a minimal output.

7

Comparing Three Files

Use the program `diff3` to compare three files and show any differences among them. (`diff3` can also merge files; see “Merging from a Common Ancestor” on page 271). 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 “Invoking the `diff3` Utility” on page 299 for more information on how to run `diff3`. The following documentation discusses comparing files.

- “A Third Sample Input File” (below)
- “Detailed Description of `diff3` Normal Format” on page 268
- “`diff3` Hunks” on page 269
- “An Example of `diff3` Normal Format” on page 269

A Third Sample Input File

The following is a third sample file (`tao`) 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 “Two Sample Input Files” on page 244).

```
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

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 the origin of the lines.

Normally, two spaces precede each copy of an input line to distinguish it from the commands. But with the `-T` or `--initial-tab` options, `diff3` uses a tab instead of two spaces; this lines up tabs correctly. See “Preserving Tabstop Alignment” on page 263 for more information.

Commands take the following forms.

*file: l*_a

This hunk appears after line, *l*, of file, *file*, containing 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, `1:11` means that the hunk follows line 11 in the first file and contains no lines from that file.

file: rc

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, it is distinguished on output from a full line by a following line that starts with `\` (see “Incomplete Lines” on page 315).

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 “Hunks” on page 238). 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 then have the following output:

```
====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, is then a three-way hunk containing the last line of each file.

An Example of diff3 Normal Format

Here is the output of the command, `diff3 lao tzu tao` (see “A Third Sample Input File” on page 268 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
```

8

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 (`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`.

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, such as 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 options, `-A` or `--show-all` (see “Marking Conflicts” on page 273). The following is the output of the command, `diff3`, with each of these three options (see “A Third Sample Input File” on page 268 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
.
```

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` options act 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 “A Third Sample Input File” on page 268), `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` options output less information than the `-A` or `--show-all` options, because the output is only unmerged changes, and never output of the contents of the second file. 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 three-way overlapping changes, 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` options to show alternate names in the `<<<<<<<`, `|||||`, and `>>>>>>` brackets. This option can be given up to three times, once for each input file. `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, `A`, `B`, and `C`.

Generating the Merged Output Directly

With the `-m` or `--merge` options, `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`, would have the following output (see “A Third Sample Input File” on page 268 for a copy of the input files):

```
<<<<<<< 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
```

How `diff3` Merges Incomplete Lines

With `-m`, incomplete lines (see “Incomplete Lines” on page 315) are simply copied to the output as they are found; if the merged output ends in a 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.

Saving the Changed File

Traditional Unix `diff3` generates an `ed` script without the trailing `w` 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`.

9

`sdiff` Interactive Merging

With `sdiff`, you can merge two files interactively based on a side-by-side format comparison with `-y` (see “Showing Differences Side by Side” on page 251). Use `-o file` or `--output=file`, to specify where to put the merged text. See “Invoking the `sdiff` Utility” on page 311 for more details on the options to `sdiff`. Another way to merge files interactively is to use the Emacs Lisp package, `emerge`. See “Merging Files with Emerge” in *The GNU Emacs Manual*[†] for more information.

[†] The GNU Emacs Manual is published by the Free Software Foundation (ISBN 1-882114-03-5).

Specifying `diff` Options to the `sdiff` Utility

The following `sdiff` options have the same meaning as for `diff`. See “diff Options” on page 292 for the use of these options.

```
-a -b -d -i -t -v
-B -H -I regex

--ignore-blank-lines --ignore-case
--ignore-matching-lines= regex --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 `sdiff`'s `-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` options (see “Showing Differences Side by Side” on page 251).

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 using the **Enter** key.

`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.

10

Merging with the `patch` Utility

`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 ensure 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 “Hunks” on page 238) 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 “Invoking the patch Utility” on page 303 for detailed information on the options to patch. See “Backup File Names” on page 304 for more information on how patch names backup files. See “Naming Reject Files” on page 305 for more information on where patch puts reject hunks.

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 shown in the following discussion are the only ones that `patch` can understand.

```
-c
--context
    Context diff.
-e
--ed
    ed script.
-n
--normal
    Normal diff.
-u
--unified'
    Unified diff.
```

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.

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` options, making `patch` compare white space loosely so that any sequence of white space in the patch file matches any sequence of whitespace in the input files. Non-whitespace characters must still match exactly. Each line of the context must still match a line in the input file.

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` options. `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 use the keystroke sequence, `Ctrl-c`, to kill the `patch` process.

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` options set the maximum fuzz factor to `lines`. This option only applies to context and unified `diffs`; it ignores up to `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 “Naming Reject Files” on page 305 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.

Removing Empty Files

Sometimes when comparing two directories, the first directory contains a file that the second directory does not. If you give `diff` a `-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` options to `patch` delete output files that are empty after applying the `diff`.

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 “Tips for Making Distributions with Patches” on page 287 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`.

Messages and Questions from the `patch` Utility

`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` options. There are two ways you can prevent `patch` from asking you any questions. The `-f` or `--force` options assume 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.

11

Tips for Making Distributions with Patches

The following discussions detail 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` options, as needed. Avoid sending 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.

12

Invoking the `cmp` Utility

The `cmp` command compares two files, and if they differ, tells the first byte and line number where they differ.

Its arguments are: `cmp options ... from-file [to-file]`.

The file name with `-` 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.

`cmp` Options

The following 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: `-c1` is equivalent to `-c -1`.

`-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-` (“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-` (“meta”).

`--quiet`

`-s`

`--silent`

Do not print anything; return exit status indicating whether files differ.

`--verbose`

Print the (decimal) offsets and (octal) values of all differing bytes.

`-v`

`--version`

Output the version number of `cmp`.

13

Invoking the `diff` Utility

`diff options ...from-file to-file` is the format for running the `diff` command.

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` options are 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.

diff Options

The following 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 “Context Format” on page 246) or `-u` (see “Unified Format” on page 248). 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 “Binary Files and Forcing Text Comparisons” on page 241.

`-b`

Ignore changes in amount of white space. See “Suppressing Differences in Blank and Tab Spacing” on page 239.

`-B`

Ignore changes that just insert or delete blank lines. See “Suppressing Differences in Blank Lines” on page 239.

`--binary`

Read and write data in binary mode. See “Binary Files and Forcing Text Comparisons” on page 241.

`--brief`

Report only whether the files differ, not the details of the differences. See “Summarizing Which Files Differ” on page 240.

`-c`

Use the context output format. See “Context Format” on page 246.

`-C lines`

`--context[=lines]`

Use the context output format, showing *lines* (an integer) lines of context, or

three if *lines* is not given. See “Context Format” on page 246. 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 “Line Group Formats” on page 255.

`-d`

Change the algorithm perhaps find a smaller set of changes. This makes `diff` slower (sometimes much slower). See “diff Performance Tradeoffs” on page 265.

`-D name`

Make merged `#ifdef` format output, conditional on the pre-processor macro *name*. See “Merging Files with If-then-else” on page 255.

`-e`

`--ed`

Make output that is a valid `ed` script. See “ed Scripts” on page 252.

`--exclude=pattern`

When comparing directories, ignore files and subdirectories whose basenames match *pattern*. See “Comparing Directories” on page 261.

`--exclude-from=file`

When comparing directories, ignore files and subdirectories whose basenames match any pattern contained in *file*. See “Comparing Directories” on page 261.

`--expand-tabs`

Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See “Preserving Tabstop Alignment” on page 263.

`-f`

Make output that looks vaguely like an `ed` script but has changes in the order they appear in the file. See “Forward ed Scripts” on page 254.

`-F regexp`

In context and unified format, for each hunk of differences, show some of the last preceding line that matches *regexp*. See “Suppressing Lines Matching a Regular Expression” on page 240.

`--forward-ed`

Make output that looks vaguely like an `ed` script but has changes in the order they appear in the file. See “Forward ed Scripts” on page 254.

`-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 “diff Performance Tradeoffs” on page 265.

`--horizon-lines=lines`

Do not discard the last *lines* lines of the common prefix and the first *lines* lines of the common suffix. See “diff Performance Tradeoffs” on page 265.

- i
Ignore changes in case; consider uppercase and lowercase letters equivalent. See “Suppressing Case Differences” on page 240.
- I *regexp*
Ignore changes that just insert or delete lines that match *regexp*. See “Suppressing Lines Matching a Regular Expression” on page 240.
- ifdef=*name*
Make merged if-then-else output using *name*. See “Merging Files with If-then-else” on page 255.
- ignore-all-space
Ignore white space when comparing lines. See “Suppressing Differences in Blank and Tab Spacing” on page 239.
- ignore-blank-lines
Ignore changes that just insert or delete blank lines. See “Suppressing Differences in Blank Lines” on page 239.
- ignore-case
Ignore changes in case; consider upper- and lower-case to be the same. See “Suppressing Case Differences” on page 240.
- ignore-matching-lines=*regexp*
Ignore changes that just insert or delete lines that match *regexp*. See “Suppressing Lines Matching a Regular Expression” on page 240.
- ignore-space-change
Ignore changes in amount of white space. See “Suppressing Differences in Blank and Tab Spacing” on page 239.
- 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 “Preserving Tabstop Alignment” on page 263
- l
Pass the output through `pr` to paginate it. See “Paginating diff Output” on page 264.
- L *label*
Use *label* instead of the file name in the context format (see “Detailed Description of Context Format” on page 246) and unified format (see “Detailed Description of Unified Format” on page 248) headers. See “RCS Scripts” on page 254.
- label=*label*
Use *label* instead of the file name in the context format (see “Detailed Description of Context Format” on page 246) and unified format (see “Detailed Description of Unified Format” on page 248) headers.

-
- `--left-column`
Print only the left column of two common lines in side by side format. See “Controlling Side by Side Format” on page 252.
 - `--line-format=format`
Use *format* to output all input lines in if-then-else format. See “Line Formats” on page 258.
 - `--minimal`
Change the algorithm to perhaps find a smaller set of changes. This makes `diff` slower (sometimes much slower). See “diff Performance Tradeoffs” on page 265.
 - `-n`
Output RCS-format diffs; like `-f` except that each command specifies the number of lines affected. See “RCS Scripts” on page 254.
 - `-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 “Comparing Directories” on page 261.
 - `--new-group-format=format`
Use *format* to output a group of lines taken from just the second file in if-then-else format. See “Line Group Formats” on page 255.
 - `--new-line-format=format`
Use *format* to output a line taken from just the second file in if-then-else format. See “Line Formats” on page 258.
 - `--old-group-format=format`
Use *format* to output a group of lines taken from just the first file in if-then-else format. See “Line Group Formats” on page 255.
 - `--old-line-format=format`
Use *format* to output a line taken from just the first file in if-then-else format. See “Line Formats” on page 258.
 - `-p`
Show which C function each change is in. See “Showing C Function Headings” on page 250.
 - `-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 “Comparing Directories” on page 261.
 - `--paginate`
Pass the output through `pr` to paginate it. See “Paginating diff Output” on page 264.
 - `-q`
Report only whether the files differ, not the details of the differences. See “Summarizing Which Files Differ” on page 240.

- `-r`
When comparing directories, recursively compare any sub-directories found. See “Comparing Directories” on page 261.
- `--rcs`
Output RCS-format diffs; like `-f` except that each command specifies the number of lines affected. See “RCS Scripts” on page 254.
- `--recursive`
When comparing directories, recursively compare any sub-directories found. See “Comparing Directories” on page 261.
- `--report-identical-files`
Report when two files are the same. See “Comparing Directories” on page 261.
- `-s`
Report when two files are the same. See “Comparing Directories” on page 261.
- `-S file`
When comparing directories, start with the file, *file*. This is used for resuming an aborted comparison. See “Comparing Directories” on page 261.
- `--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 “Showing C Function Headings” on page 250.
- `--show-function-line=regex`
In context and unified format, for each hunk of differences, show some of the last preceding line that matches *regex*. See “Suppressing Lines Matching a Regular Expression” on page 240.
- `--side-by-side`
Use the side by side output format. See “Controlling Side by Side Format” on page 252.
- `--speed-large-files`
Use heuristics to speed handling of large files that have numerous scattered small changes. See “diff Performance Tradeoffs” on page 265.
- `--starting-file=file`
When comparing directories, start with the file, *file*. This is used for resuming an aborted comparison. See “Comparing Directories” on page 261.
- `--suppress-common-lines`
Do not print common lines in side by side format. See “Controlling Side by Side Format” on page 252.
- `-t`
Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See “Preserving Tabstop Alignment” on page 263.

-
- 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 “Preserving Tabstop Alignment” on page 263.
 - text
Treat all files as text and compare them line-by-line, even if they do not appear to be text. See “Binary Files and Forcing Text Comparisons” on page 241.
 - u
Use the unified output format. See “Unified Format” on page 248.
 - unchanged-group-format=*format*
Use *format* to output a group of common lines taken from both files in if-then-else format. See “Line Group Formats” on page 255.
 - unchanged-line-format=*format*
Use *format* to output a line common to both files in if-then-else format. See “Line Formats” on page 258.
 - 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 “Comparing Directories” on page 261.
 - U *lines*
--unified[= *lines*]
Use the unified output format, showing *lines* (an integer) lines of context, or three if *lines* is not given. See “Unified Format” on page 248. 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 “Suppressing Differences in Blank and Tab Spacing” on page 239.
 - W *columns*
--width=*columns*
Use an output width of *columns* in side by side format. See “Controlling Side by Side Format” on page 252.
 - x *pattern*
When comparing directories, ignore files and subdirectories whose basenames match *pattern*. See “Comparing Directories” on page 261.
 - X *file*
When comparing directories, ignore files and subdirectories whose basenames match any pattern contained in *file*. See “Comparing Directories” on page 261.

`-y`

Use the side by side output format. See “Controlling Side by Side Format” on page 252.

14

Invoking the `diff3` Utility

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.

`diff3` Options

The following 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 “Binary Files and Forcing Text Comparisons” on page 241.

`-A`

Incorporate all changes from older to yours into mine, surrounding all conflicts with bracket lines. See “Marking Conflicts” on page 273.

`-e`

Generate an `ed` script that incorporates all the changes from older to yours into mine. See “Selecting Which Changes to Incorporate” on page 272.

`-E`

Like `-e`, except bracket lines from overlapping changes in first and third files. See “Marking Conflicts” on page 273. 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 “Selecting Which Changes to Incorporate” on page 272.

`--easy-only`

Like `-e`, except output only the non-overlapping changes. See “Selecting Which Changes to Incorporate” on page 272.

`-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 “Saving the Changed File” on page 275.

`--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 “Preserving Tabstop Alignment” on page 263.

`-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 “Marking Conflicts” on page 273.

- 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 “Generating the Merged Output Directly” on page 274.
- overlap-only
Like `-e`, except output only the overlapping changes. See “Selecting Which Changes to Incorporate” on page 272.
- show-all
Incorporate all unmerged changes from *older* to *yours* into *mine*, surrounding all overlapping changes with bracket lines. See “Marking Conflicts” on page 273.
- show-overlap
Like `-e`, except bracket lines from overlapping changes in first and third files. See “Marking Conflicts” on page 273.
- 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 “Preserving Tabstop Alignment” on page 263.
- text
Treat all files as text and compare them line-by-line, even if they do not appear to be text. See “Binary Files and Forcing Text Comparisons” on page 241.
- v
- version
Output the version number of `diff3`.
- x
Like `-e`, except output only the overlapping changes. See “Selecting Which Changes to Incorporate” on page 272.
- X
Like `-E`, except output only the overlapping changes. In other words, like `-x`, except bracket changes as in `-E`. See “Marking Conflicts” on page 273.
- 3
Like `-e`, except output only the nonoverlapping changes. See “Selecting Which Changes to Incorporate” on page 272.

15

Invoking the `patch` Utility

Normally `patch` is invoked using: `patch < patchfile`.

The full format for invoking `patch` is a declaration like the following example.

```
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 “Multiple Patches in a File” on page 284 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 “Backup File Names” on page Backup File Names 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.

Applying Patches in Other Directories

The `-d directory` or `--directory=directory` options to `patch` make *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 the following example:

```
| 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]` options 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/new`. Using `-p` or `-p0` gives the entire file name unmodified, `-p1` gives `gnu/src/emacs/etc/new` (no leading slash), `-p4` gives `etc/news`, and not specifying `-p` at all gives `news`.

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` options use *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 “Transposing Text” in *The GNU Emacs Manual*[†], for more information on backup versions in Emacs). They also recognize synonyms that are more descriptive. The valid values are listed in the following; 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.

Naming Reject Files

The names for reject files (files containing patches that `patch` could not find a place to

[†] The Free Software Foundation publishes *The GNU Emacs Manual* (ISBN 1-882114-03-5).

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.

patch Options

The following summarizes 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 “Backup File Names” on page 304.

`-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 “Backup File Names” on page 304.

`--batch`

Do not ask any questions. See “Messages and Questions from the `patch` Utility” on page 285.

`-c`

`--context`

Interpret the `patch` file as a context `diff`. See “Selecting the `patch` Input Format” on page 282.

`-d directory`

`--directory=directory`

Makes `directory` the current directory for interpreting both file names in the `patch` file, and file names given as arguments to other options. See “Applying Patches in Other Directories” on page 304.

`-D name`

Make merged if-then-else output using format. See “Merging Files with If-then-else” on page 255.

`--debug=number`

Set internal debugging flags. Of interest only to `patch` patchers.

`-e`

`--ed`

Interpret the `patch` file as an `ed` script. See “Selecting the `patch` Input Format” on page 282.

-
- E
Remove output files that are empty after the patches have been applied. See “Removing Empty Files” on page 284.
 - f
Assume that the user knows exactly what he or she is doing, and ask no questions. See “Messages and Questions from the patch Utility” on page 285.
 - F *lines*
Set the maximum fuzz factor to *lines*. See “Helping patch Find Inexact Matches” on page 283.
 - force
Assume that the user knows exactly what he or she is doing, and ask no questions. See “Messages and Questions from the patch Utility” on page 285.
 - forward
Ignore patches that `patch` thinks are reversed or already applied. See also `-R`. See “Applying Reversed Patches” on page 283.
 - fuzz=*lines*
Set the maximum fuzz factor to *lines*. See “Helping patch Find Inexact Matches” on page 283.
 - help
Print a summary of the options that `patch` recognizes, then exit.
 - ifdef= *name*
Make merged if-then-else output using format. See “Merging Files with If-then-else” on page 255.
 - 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 “Applying Patches with Changed White Space” on page 282.
 - n
 - normal
Interpret the `patch` file as a normal diff. See “Selecting the patch Input Format” on page 282.
 - N
Ignore patches that `patch` thinks are reversed or already applied. See also `-R`. See “Applying Reversed Patches” on page 283.
 - o *output-file*
 - output=*output-file*
Use *output-file* as the output file name.
 - p[*number*]
Set the file name strip count to *number*. See “Applying Patches in Other Directories” on page 304.

- `--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 “Backup File Names” on page 304.
- `--quiet`
Work silently unless an error occurs. See “Messages and Questions from the patch Utility” on page 285.
- `-r reject-file`
Use *reject-file* as the reject file name. See “Naming Reject Files” on page 305.
- `-R`
Assume that this patch was created with the old and new files swapped. See “Applying Reversed Patches” on page 283.
- `--reject-file=reject-file`
Use *reject-file* as the reject file name. See “Naming Reject Files” on page 305.
- `--remove-empty-files`
Remove output files that are empty after the patches have been applied. See “Removing Empty Files” on page 284.
- `--reverse`
Assume that this patch was created with the old and new files swapped. See “Applying Reversed Patches” on page 283.
- `-s`
Work silently unless an error occurs. See “Messages and Questions from the patch Utility” on page 285.
- `-S`
Ignore this patch from the patch file, but continue looking for the next patch in the file. See “Multiple Patches in a File” on page 284.
- `--silent`
Work silently unless an error occurs. See “Messages and Questions from the patch Utility” on page 285.
- `--skip`
Ignore this patch from the `patch` file, but continue looking for the next patch in the file. See “Multiple Patches in a File” on page 284.
- `--strip[=number]`
Set the file name strip count to *number*. See “Applying Patches in Other Directories” on page 304.
- `--suffix=backup-suffix`
Use *backup-suffix* as the backup extension instead of `.orig` or `~`. See “Backup File Names” on page 304.
- `-t`
Do not ask any questions. See “Messages and Questions from the patch Utility” on page 285.

- u
- unified
Interpret the patch file as a unified diff. See “Selecting the patch Input Format” on page 282.
- v
Output the revision header and patch level of patch.
- V *backup-style*
Select the kind of backups to make. See “Backup File Names” on page 304.
- version
Output the revision header and patch level of `patch`, then exit.
- version=control=*backup-style*
Select the kind of backups to make. See “Backup File Names” on page 304.
- x *number*
Set internal debugging flags. Of interest only to `patch` patchers.

16

Invoking the `sdiff` Utility

The `sdiff` command merges two files and interactively outputs the results.

Its arguments are: `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.

`sdiff` Options

The following 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 “Binary Files and Forcing Text Comparisons” on page 241.
- `-b`
Ignore changes in amount of whitespace. See “Suppressing Differences in Blank and Tab Spacing” on page 239.
- `-B`
Ignore changes that just insert or delete blank lines. See “Suppressing Differences in Blank Lines” on page 239.
- `-d`
Change the algorithm to perhaps find a smaller set of changes. This makes `sdiff` slower (sometimes much slower). See “diff Performance Tradeoffs” on page 265.
- `-H`
Use heuristics to speed handling of large files that have numerous scattered small changes. See “diff Performance Tradeoffs” on page 265.
- `--expand-tabs`
Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See “Preserving Tabstop Alignment” on page 263.
- `-i`
Ignore changes in case; consider uppercase and lowercase to be the same. See “Suppressing Case Differences” on page 240.
- `-I regexp`
Ignore changes that just insert or delete lines that match *regexp*. See “Suppressing Lines Matching a Regular Expression” on page 240.
- `--ignore-all-space`
Ignore white space when comparing lines. See “Suppressing Differences in Blank and Tab Spacing” on page 239.
- `--ignore-blank-lines`
Ignore changes that just insert or delete blank lines. See “Suppressing Differences in Blank Lines” on page 239.
- `--ignore-case`
Ignore changes in case; consider uppercase and lowercase to be the same. See “Suppressing Case Differences” on page 240.

-
- `--ignore-matching-lines=regexp`
Ignore changes that just insert or delete lines that match *regexp*. See “Suppressing Lines Matching a Regular Expression” on page 240.
 - `--ignore-space-change`
Ignore changes in amount of whitespace. See “Suppressing Differences in Blank and Tab Spacing” on page 239.
 - `-l`
 - `--left-column`
Print only the left column of two common lines. See “Controlling Side by Side Format” on page 252.
 - `--minimal`
Change the algorithm to perhaps find a smaller set of changes. This makes `sdiff` slower (sometimes much slower). See “diff Performance Tradeoffs” on page 265.
 - `-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 “Controlling Side by Side Format” on page 252.
 - `--speed-large-files`
Use heuristics to speed handling of large files that have numerous scattered small changes. See “diff Performance Tradeoffs” on page 265.
 - `-t`
Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See “Preserving Tabstop Alignment” on page 263.
 - `--text`
Treat all files as text and compare them line-by-line, even if they do not appear to be text. See “Binary Files and Forcing Text Comparisons” on page 241.
 - `-v`
 - `--version`
Output the version number of `sdiff`.
 - `-w columns`
 - `--width=columns`
Use an output width of *columns*. See “Controlling Side by Side Format” on page 252. For historical reasons, this option is `-W` in `diff`, `-w` in `sdiff`.
 - `-W`
Ignore horizontal white space when comparing lines. See “Suppressing Differences in Blank and Tab Spacing” on page 239. For historical reasons, this option is `-w` in `diff`, `-W` in `sdiff`.

17

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 “Suppressing Differences in Blank and Tab Spacing” on page 239).

An incomplete line is normally distinguished on output from a full line by a following line that starts with `\`. However, the RCS format (see “RCS Scripts” on page 254) 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 “Controlling Side by Side Format” on page 252. The if-then-else line format preserves a line’s incompleteness with `%L`, and discards the new-line with `%1`; see “Line Formats” on page 258. Finally, with the `ed` and forward `ed` output formats (see “diff Output Formats” on page 243) `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
.
```

18

Future Projects for `diff` and `patch` Utilities

The following discussions have 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.

Suggested Projects for Improving GNU `diff` and `patch` Utilities

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.

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.

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.

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.

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.

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.

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 the first file corresponds to `bar` in the second file, 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.

Reporting Bugs

If you think you have found a bug in GNU `cmp`, `diff`, `diff3`, `sdiff`, or `patch`, 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.

Index

Symbols

!, indicator character 247

#, for comments 89

#else directive 255

#endif directive 255

#ifdef directive 255

#ifndef directive 255

\$ 185, 223

\$\$ variables 222

\$\$, automatic variable 184

\$(186, 186, 223, 223

\$(%D) variables 223

\$(%D) variants 186

\$(%F) variables 223

\$(%F) variants 186

\$(*D) variables 223

\$(*D) variants 186

\$(*F) variables 223

\$(*F) variants 186

\$(+D) variables 223

\$(+F) variables 223

\$(?D) variables 223

\$(?D) variants 186

\$(?F) variables 223

\$(?F) variants 186

\$(@D) variables 223

\$(@D) variants 186

\$(@F) variables 223

\$(@F) variants 186

\$(*D) variables 223

\$(*D) variants 186

\$(*F) variables 223

\$(*F) variants 186

\$(addprefix prefix, names...) 152

- \$(addsuffix suffix, names...) 152
- \$(basename names...) 152
- \$(dir names...) 152
- \$(filter pattern ...,text) 150
- \$(filter-out pattern ...,text) 151
- \$(findstring find,in) 150
- \$(firstword names...) 153
- \$(join list1, list2) 153
- \$(notdir names ...) 152
- \$(patsubst pattern, replacement, text) 149
- \$(sort list) 151
- \$(strip string) 150
- \$(subst from, to, text) 149
- \$(suffix names ...) 152
- \$(wildcard pattern) 153
- \$(word n, text) 153
- \$(words text) 153
- \$* variables 223
- *, automatic variable 185
- \$+ variables 223
- +, automatic variable 185
- ? variables 223
- ?, automatic variable 185
- @ variables 222
- @, automatic variable 184
- @, for varying commands 106
- ^ variables 223
- ~, automatic variable 185
- \$ORIGIN processing 24
- % 257
- % character 98, 107
- % in pattern rules 182
- %%, for line group formatting 257, 259
- %, for conversion specifications 257
- %=, for line group formatting 257
- %>, for line group formatting 257
- %c' C', for line group formatting 257, 259
- %c', for line group formatting 259
- %c:' 259
- %c'' 259
- %c'O', for line group formatting 257
- %d, for line group formatting 257
- %L, for line group formatting 258
- %l, for line group formatting 258
- %o, for line group formatting 257
- %X, for line group formatting 257
- %x, for line group formatting 257
- (markers 251
-) markers 251
- *, comments 72
- + as prefix character 124
- + prefix 145
- +, adding a textline 249
- +, indicator character 247
- += syntax 199
- +=, appending values to variables 137
- , deleting a text line 249
- , for specifying left-justification 257
- , indicator character 247
- , prefix character 124
- ., indicator of end of output in ed 253
- .bss output section 32
- .c file 174
- .data output section 32
- .DEFAULT 104, 189, 192
- .exe suffix 17
- .EXPORT_ALL_VARIABLES 105, 121
- .IGNORE 104
- .PHONY 104, 199
- .POSIX 123
- .PRECIOUS 104, 119
- .SILENT 104
- .SUFFIXES 104, 169, 175, 190
- .text 32
- .text section 32
- / markers 251
- /lib 20
- /usr/lib 20
- :=, for simply expanded variables 135
- < markers 251
- <<<<<<<, marker 272
- =, equals sign 8
- =, for setting variable values 135
- load-average 169
- max-load 169
- > markers 251

>>>>>>, marker 272
 @ as prefix character 124
 @@, line in unified format 250
 __.SYMDEF 195
 __main 44
 |markers 251

A

-A 64
 -a 242
 a.out 6, 13, 68
 a.out object file format 71
 -Architecture 9, 64
 aborting execution of make 117
 ABSOLUTE 72
 absolute address 64
 absolute addresses 72
 absolute locations of memory 14
 abstraction payback 68
 -ACA 64
 add text lines 245
 addprefix 222
 address 48
 addresses 30
 --add-stdcall-alias 24
 addsuffix 222
 aggregates 70
 ALIAS 72
 alignment 39, 47, 68
 alignment constraints 20
 alignment information
 propagating 68
 alignment of input section 39
 all 161
 all target 208
 allocatable 30
 alternate file names 250
 ar program 195
 AR variable 179
 arbitrary sections, supported 20
 architecture-independent files 213
 archive file symbol tables 201

archive files 5, 23, 193
 archive libraries 64
 archive member 184
 archive member target 191
 archive member targets 186
 archive suffix rules 196
 archive-maintaining program 181
 archives, specifying 15
 ARFLAGS variable 181
 AS variable 180
 ASFLAGS variable 181
 assembler 181
 assembler programs 176
 -assert 15
 assignment operators 35
 --assume-new= option 163
 --assume-new=file option 170
 --assume-old=file option 169
 AT&T
 Link Editor Command Language syntax 5
 authors, listing 170
 automatic 155
 automatic variable 155, 198
 automatic variables 100, 128, 134, 135, 184,
 222
 --auxiliary 10

B

-b 9, 239
 -b , or -format 27
 -b option 167
 b.out 68
 backslashes 149
 backslashes (99
 backslash-newline 135
 backup file names 304
 bal 64
 balx 64
 BASE 72
 --base-file 24
 basename 222
 bash script 207

- Bdynamic 15
 - BFD 27, 67, 69
 - abstraction payback 68
 - archive files 67
 - back end conversions 68
 - back end, symbol table 70
 - canonical relocation record 70
 - coff-m68k variant 72
 - conversion and output, information loss 68
 - converting between formats 68
 - files 69
 - internal canonical form of external formats 69
 - object and archive files 67
 - object file format 67
 - optimizing algorithms 68
 - section 69
 - subroutines 68
 - BFD libraries 9, 18, 67
 - big endian 69
 - big endian COFF 69
 - binary and text file comparison 241
 - binary format for output object file 18
 - binary input files 27
 - binary input files, specifying 8
 - bindir 213
 - Bison 83, 209, 211
 - BISONFLAGS 211
 - blank and tab difference suppression 239
 - blank lines 239
 - bra 64
 - braces, unmatched 148
 - brief difference reports 240
 - brief option 240
 - Broken pipe signal 116
 - BSD 198, 212
 - BSD make 202
 - Bshareable 20
 - bsr 64
 - Bstatic 15
 - Bsymbolic 16
 - buffer.h 81
 - bugs, reporting 203
 - built-in expansion functions 151
 - built-in implicit rule, overriding 188
 - built-in implicit rules 90, 174, 179, 200
 - built-in rules 179
- ## C
- c 10, 71
 - C compilation 187
 - C function headings, showing 250
 - C if-then-else output format 255
 - C option 167
 - C option, multiple specifications 167
 - C preprocessor 176, 199
 - C programs 176
 - C++ mangled symbol names 16
 - C++ programs 176
 - C++ programs, linking 14
 - C, C++, Prolog regular expressions 250
 - cal 64
 - call-shared 15
 - calx 64
 - canned sequence 125
 - canned sequence of commands 124
 - canned sequence with the define directive 124
 - canonical format, destination format 69
 - canonicalization 69
 - case difference suppression 240
 - catalogue of predefined implicit rules 175
 - CC variable 180
 - CFLAGS 129, 136, 138, 164, 174, 186, 211
 - CFLAGS variable 181
 - chain, defined 181
 - change commands 245, 253
 - change compared files 252
 - changed-group-format= 256
 - check 162
 - check target 210
 - check-sections 16
 - child processes 116
 - CHIP 72
 - clean 80, 161
 - clean target 209
 - clobber 161

-
- cmp 242
 - cmp command options 289
 - CO variable 180
 - COFF 6
 - COFF debuggers 70
 - COFF files, sections 68
 - COFF object file format 20
 - COFF section names 68
 - coff-m68k 72
 - COFLAGS variable 181
 - COLLECT_NO_DEMANGLE 17, 27
 - columnar output 252
 - command 81
 - command line 89, 155
 - command line override 250
 - command sequence, defining 124
 - command.h 81
 - command.o 83
 - command-line options 198
 - commands, echoing 114
 - commands, empty 125
 - commands, errors 117
 - commas 148
 - comments 31, 88, 131
 - common symbol 21
 - common symbols 41
 - common variables 70
 - comparing text files with null characters 241
 - compilation errors 165
 - compilation-and-linking 187
 - compile 118
 - COMPILE.c 179
 - COMPILE.x 179
 - compiler 44
 - compiler drivers, specifying 9
 - compiler switches 138
 - compiling 5, 176
 - compiling a program, testing 165
 - compiling C programs 97
 - complex makefile, example 227
 - complex values 136
 - computed variable names 132
 - conditional 141
 - conditional directive 121
 - conditional example 141
 - conditional execution 199
 - conditional syntax 143
 - configure 216
 - conflict 272
 - constraints 187
 - constructor 44
 - CONSTRUCTORS 43
 - constructors 13
 - contacting Red Hat iii
 - context 244
 - context format and unified format 246
 - context output format 250
 - controlling a linker 7
 - conventions for writing the Makefiles 205
 - COPY 45
 - copyright 170
 - counter relative instructions 64
 - CPP variable 180
 - CPPFLAGS 211
 - CPPFLAGS variable 181
 - CREATE_OBJECT_SYMBOLS 43
 - cref 16
 - csh script 207
 - CTANGLE variable 180
 - CWEAVE variable 180
 - CXX variable 180
 - CXXFLAGS variable 181
- ## D
- d option 168
 - d, -dc, -dp 10
 - dashes in option names 8
 - data section 30
 - data segments 13
 - datadir 213
 - debug option 168
 - debugger symbol information 14
 - debugging
 - addresses in output files 70
 - mapping between symbols 70
-

-
- relocated addresses of output files 70
 - source line numbers 70
 - default 155
 - define 220
 - define directive 134, 137
 - define directives 137
 - define option 199
 - definitions of variables 89
 - defs.h 81, 82
 - defsym 16
 - delete text lines 245
 - demand pageable bit 69
 - demand pageable bit, write protected text bit set 69
 - demangle 16
 - dependencies 82, 165, 169, 185
 - dependencies, analogous 107
 - dependencies, duplicate 223
 - dependencies, explicit 175
 - dependencies, from source files 89
 - dependencies, generating automatically 109
 - dependency 81
 - dependency loop 92
 - dependency patterns 174
 - dep-patterns 107
 - destination format, canonical format 69
 - destructors 44
 - diagnostics 6
 - diff and patch, overview 235
 - diff output, example 244
 - diff sample files 244
 - diff, older versions 244
 - diff3 242
 - diff3 comparison 267
 - dir 131, 221
 - directives 88, 219
 - directories 265
 - directories, implicit rules 101
 - directory option 167, 199
 - directory search features 97
 - directory search with linker libraries 101
 - directory, rules for cleaning 86
 - directory, specifying 168
 - disable-new-dtags 23
 - disable-stdcall-fixup 25
 - discard-all 14
 - discard-locals 15
 - displaying files to open 21
 - dist 162, 228
 - dist target 210
 - distclean 161
 - distclean target 209
 - dlldump() 24
 - dll 24
 - DLLs 25
 - DLLs (Dynamic Link Libraries) 6
 - dlltool 24
 - dlopen() 24
 - dn 15
 - documentation 215
 - dollar signs 94, 128
 - double-colon rule 91, 175
 - double-colon rules 109
 - double-suffix 189
 - double-suffix rule 189
 - drivers 9
 - dry-run option 162, 169
 - DSECT 45
 - DSO (Dynamic Shared Object) files 24
 - DT_AUXILIARY 11
 - DT_FILTER 11
 - DT_INIT 17
 - DT_SONAME 12
 - dummy pattern rules 188
 - duplication 83
 - dvi target 210
 - dy 15
 - dynamic linker 17
 - Dynamic Shared Object (DSO) files 24
 - dynamic-linker 17
- ## E
- e 10
 - e flag, for exiting 110
 - e option 168
 - EB 10
-

echo command 125
 echoing 114
 ed output format 253
 ed script example 254
 ed script output format 252
 edit 81, 82
 editor 80
 editor, recompiling 80
 EFL programs 202
 -EL 10
 ELF executables 17, 19
 ELF platforms 16
 ELF systems 19
 else 142, 220
 Emacs' compile command 118
 Emacs' M-x compile command 166
 --embedded-relocs 17
 empty command 114
 empty commands 92
 empty files 284
 empty strings 152
 empty target 103
 emulation linker 12
 --enable-new-dtags 23
 --enable-stdcall-fixup 25
 END 72
 undef 124, 138, 220
 --end-group 15
 endif 142, 220
 entry command-line option 32
 entry point 32
 --entry= 10
 environment 155
 environment override 155
 environment variable 26, 90
 environment variables 138
 environment, redefining 156
 --environment-overrides option 168
 environments 19
 environments that override assignments 138
 error messages from make 219
 errors 6
 errors, continuing make after 168

errors, ignoring, in commands 168
 errors, tracking 170
 --exclude-symbols 25
 exec_prefix variable 212
 executable 30, 187
 executables 80
 execution of programs 10
 execution time 20
 exit status 117
 exit status of make 160
 --expand-tabs option 259
 explicit dependencies 175
 explicit rules 88
 export 220
 export directive 121, 139
 --export-all-symbols 25
 exporting variables 120
 expression 39
 extern int i 21

F

-F 10, 11
 -F option 250
 -f option 168
 -F regexp option 250
 F variants 186
 -f, or --file for naming makefile 89
 failed shell commands 165
 fatal error 90
 fatal messages 285
 fatal signal 118
 FC variable 180
 FFLAGS variable 181
 file 155
 file alignment 25
 file differences, summarizing 240
 file name endings 174
 file names 151
 file names, showing alternate 250
 file, reading 168
 --file=file option 168
 --file-alignment 25

- filename 13, 19
- files 69
- files, marking them up to date 170
- filter 11
- filter 221
- findstring 221
- fini 17
- firstword 222
- FLAGS 211
- FN, for line group formatting 259
- force 92
- FORCE_COMMON_ALLOCATION 10
- force-exe-suffix 17
- foreach 222
- foreach function 153
- FORMAT 72
- format= 9
- Fortran programs 176
- forward ed script output format 254
- from-file 247
- full lines 315
- function 31
- function call 147
- function call syntax 148
- function invocations 132
- function references 135
- functions 147
- functions, transforming text with 199

G

- G 11
- g 11
- garbage collection 17
- gc-sections 17
- generating merged output directly 274
- GET variable 180
- GFLAGS variable 181
- global constructors
 - warnings 22
- global optimizations 18
- global or static variable 31
- global symbol 35

- global symbols 16, 21
- global, static, and common variables 70
- GNU Emacs 305
- GNUTARGET 26, 27
- goals 82, 160
- GP register 11
- gp-size= 11
- GROUP 8, 61

H

- h 11
- h option 168
- headings 250
- heap 25
- hello.o 8
- help 17
- help option 168, 199
- Hitachi h8/300 18
- hunks 238, 253

I

- I 12
- i 13, 18
- I dir option 168
- i option 168
- i960 9, 18, 64
- IEEE 72
- IEEE Standard 1003.2-1992 201
- IEEE Standard 1003.2-1992 (POSIX.2) 199
- ifdef 144, 220
 - ifdef=HAVE_WAITPID option 255
- ifeq 142, 143, 220
- ifndef 144, 220
- ifneq 143, 220
- if-then-else example 260
- if-then-else format 257, 259
- if-then-else output format 255
 - ignore option 251
 - ignore-case option 240
 - ignore-errors 117
 - ignore-errors option 168
 - ignore-space-change 239

- image-base 25
- imperfect patch 282
- implicit intermediate files 198
- implicit linker script 61
- implicit rule 84, 97, 108
- implicit rule for archive member targets 194
- implicit rule search algorithm 191
- implicit rule, canceling 188
- implicit rule, cancelling or overriding 175
- implicit rule, using 174
- implicit rules 88, 174
- implicit rules, chains of 181
- implicit rules, predefined 175
- implicit rules, special 182
- include 220
- include directive 199
- include directive 89
- include-dir 90
- includedir 215
- include-dir=dir option 168
- includes 136
- incomplete lines 315
- incomplete lines, merging with diff3 275
- incremental links 12
- indicator characters 247
- INFO 45
- info target 209
- infodir 208, 214
- information loss 68
- information loss with BFD 68
- init 17
- initial values of variables 134
- initialized data 31
- INPUT 8, 61
- input files 64
- input section 30
- input section description 39, 40
- input section wildcard patterns 40
- insert.c 83
- insert.o 83
- insertions 247
- INSTALL 211
- install 162

- install target 208
- INSTALL_DATA 208, 212
- INSTALL_PROGRAM 212
- installcheck target 210
- installdirs target 210
- int i 21
- int i = 1 21
- intermediate file 181
- internal canonicals 69
- invariant text 133

J

- jmp 64
- job slots 116
- jobs= 168
- j 168
- jobs, for simultaneous commands 116
- jobs, running 169
- jobs, running simultaneously 168
- jobs, specifying number 168
- join 222
- jsr 64
- just-print option 162, 169
- just-print, or -n 91
- just-symbols= 13

K

- k option 168
- kbd.o 82, 83
- keep-going 118
- keep-going option 165, 168
- keywords 43
- keywords, unrecognized 72
- kill-at 26
- killing the compiler 118
- ksh script 207

L

- L 12, 64
- L 15
- l 12, 64

- L option 246, 251
- l option 251
- la, add text command 253
- label option 246
- last-resort implicit rule 188
- ld
 - BFD libraries for operating on object files 6
 - command-line options 8
 - compiling 5
 - configuring 12
 - configuring with default input format 9
 - controlling a linker 7
 - dynamic libraries 15
 - Link Editor Command Language syntax 5
 - Linker Command Language files 5
 - machine-dependent features 63
 - object and archive files 5
 - optimizing 17
 - shared libraries 15
 - shared link 19
 - symbol references 5
 - symbols 70
 - warning messages 72
- LD_LIBRARY_PATH 20
- LD_RUN_PATH 19
- LD_RUN_PATH 19
- LDEMULATION 27
- LDFLAGS 211
- LDFLAGS variable 181
- leaf routines 64
- left-column option 252
- Lex for C programs 177
- Lex for Ratfor programs 177
- LEX variable 180
- LFLAGS variable 181
- libc.a 8
- libdir 214
- libexecdir 213
- libraries 101
- LIBRARY 24
- library 12
- library archive, updating 196
- library dependencies 200
- library= 12
- library-path= 12
- line formats 254, 258
- line group formats 255, 256
- line number list 70
- line numbers 70
- link map 72
- linker 101, 119, 177
 - addressing 18
 - canonical form 68
 - dynamic tags, enabling 23
 - ELF format options 23
 - invoking 8
 - object file format 67
- linker commands 71
- linker script example 31
- linker script, defined 29
- linker scripts commands 34
- linker, basic concepts 30
- linking 6, 211
- linking C++ 13
- linking libraries 8
- linking, partial 13
- Lint Libraries from C, Yacc, or Lex programs 178
- Linux 212
- Linux compatibility 18
- Lisp regular expressions 250
- LIST 72
- little endian 69
- little endian COFF 69
- LMA 30, 45
- ln utilities 211
- LOAD 72, 73
- l 169
- load memory address 30
- loadable 30
- local symbols 14
- localstatedir 214
- locating shared objects 19
- location counter 36
- loop names, suffixed 64
- ltry 64

M

- M 13, 72
- m 12, 27
- m option 167
- magic numbers 13, 69
- main.c 82
- main.o 82, 110
- maintainer-clean target 209
- major-image-version 26
- major-os-version 26
- major-subsystem-version 26
- MAKE 163
- make
 - automatic variables 222
 - commands 93
 - conditional variable assignment operator 131
 - CURDIR variable 119
 - default goal of rules 94
 - directives 220
 - error messages 219, 224
 - IMPORTANT tabulation 81
 - multi-processing for MS-DOS 116
 - pattern-specific variable values 140
 - recursively expanded variables 129
 - simply expanded variables 130
 - stem 107
 - TAB character 225
 - targets 93
 - target-specific variable values 139
 - text manipulation functions 221
 - variables 223
- make commands, recursive 119
- make options 162
- MAKE variable 120
- MAKE variables 224
- make with no arguments 159
- make, invoking recursively 139
- make, modifying 171
- make, running 159
- make, version number 200
- MAKE_VERSION 200
- makefile 168
 - Makefile commands 207
 - makefile, defined 80
 - makefile, example 227
 - makefile, naming 88, 160
 - makefile, overriding another makefile 92
 - makefile, using variables for object files 84
 - makefile=file option 168
 - MAKEFILES 90, 120
 - MAKEFILES variables 223
 - makefiles, debugging 171
 - makefiles, portable 197
 - makefiles, what they contain 88
 - MAKEFLAGS 120, 122, 145, 170
 - MAKEFLAGS variables 224
 - MAKEINFO variable 180
 - MAKELEVEL 121, 130, 200
 - MAKELEVEL variables 224
 - MAKEOVERRIDES 122
 - man1dir 215
 - man1ext 216
 - man2dir 215
 - man2ext 216
 - mandir 215
 - manext 215
 - Map 17
 - mark conflicts 273
 - markers 251
 - match-anything pattern rule 189
 - match-anything rules 187
 - matching 177
 - max-load option 117
 - member name 185
 - MEMORY 8
 - memory descriptor 68
 - memory, reserve 25, 26
 - memory, symbolic 14
 - merge commands 278
 - merged output format 255
 - merged output with diff3 274
 - merging two files 255
 - messages 285
 - minimal 239
 - minimal option 265

- minor-image-version 26
- minor-os-version 26
- minor-subsystem-version 26
- mk program 198, 199
- modified references 132
- Modula-2 programs 176
- mostlyclean 161
- mostlyclean target 209
- mov.b instructions 64
- MRI 10
 - a.out object file format 71
 - c 10
 - script files 10
 - T 10
- mri-script= 10
- multiple -C options 167
- multiple members in a archive file 199
- multiple patches 284
- multiple targets 105
- mv utilities 211

N

- N 13
- n 13
- n flag 170
- n option 162, 169, 254
- NAME 73
- name patterns 89
- names of files, alternates 250
- nested references 134
- nested variable reference 132
- nests of recursive make commands 170
- new jobs commands 169
- new-file= option 163
- new-file=file option 170
- new-group-format= 256
- NMAGIC 13
- nmagic 13
- no-builtin-rules option 169, 175
- no-check-sections 16
- NOCROSSREFS 47
- no-demangle 16

- no-gc-sections 17
- noinhibit-exec 18
- no-keep-going option 170
- no-keep-memory 17
- NOLOAD 45
- non_shared 15
- non-fatal messages 285
- non-text files 241
- no-print-directory option 170
- no-print-directory, disabling 124
- normal diff output format 244
- notdir 221
- no-undefined 18
- no-warn-mismatch 18
- no-whole-archive 18

O

- O 13
- o 13, 73
- o file option 169
- Oasys 69
- objdump 18, 31, 67
- object and archive files 67
- object file 30, 80, 175
- object file format 30, 67
- object file formats 34, 44
- object file names 84
- object files 5
- object files, required 23
- object formats, alternatives, supported 9
- objects 84
- offormat 18
- offormat srec, --offormat=sre 9
- offormat, -offormat 8
- old files 169
- old-fashioned suffix rule 196
- old-file option 199
- old-file=file option 169
- old-group-format= 256
- oldincludedir 215
- OMAGIC 13
- omagic 13

opened input files 21
 option 168, 169
 options
 repeating 8
 options variable 211
 options with machine dependent effects 18
 ORDER 73
 origin 155, 222
 origin function 155
 OUTPUT 13
 output section 30, 38, 42
 output section address 39
 output, understanding easier 123
 OUTPUT_FORMAT 18, 72
 OUTPUT_OPTION 179
 --output-def 26
 output-name 73
 overlap 272
 overlapping contents comparison 247
 OVERLAY 45, 48
 overlay description 47
 overridden by command line argument 164
 override 155, 220
 override define 220
 override directive 137
 override directives with define directives 137
 overview 235

P

-p option 169
 page-numbered and time-stamped output 264
 —paginate option 251
 paginating diff output 264
 parallel execution 117, 195
 parallelism 198
 parentheses 194
 parentheses, unmatched 148
 Pascal compilation 187
 Pascal programs 176
 passing down variables 120
 patch, merging with 281
 patches in another directory 304

patches with whitespace 283
 patsubst 132, 199, 221
 pattern matches 187
 pattern rule 108, 182, 191
 pattern rules 89, 198
 pattern rules, examples 183
 pattern rules, writing 169
 PC variable 180
 PE linker 24
 performance of diff 265
 PFLAGS variable 181
 phdr 46
 phony target 101
 phony targets 199
 POSIX.2 standard 201
 Posix-compliant systems, comparing 241
 predefined implicit rules 175
 predefined rules 175
 prefix variable 212
 PREPROCESS.S 179
 PREPROCESS.x 179
 preprocessor 211
 print 96, 162
 print the commands 169
 --print-data-base option 169
 --print-directory 123
 --print-directory option 170
 printf conversion specification 257
 printing 15
 printing input filenames 14
 --print-map 13
 problems iii
 processing symbol tables 68
 program-counter relative instructions 64
 PROVIDE 36
 PUBLIC 73
 punctuation 128

Q

-q option 163, 169, 240
 -qmagic 18
 question mode 169

--question option 163, 169
--question, or -q 91
questions 285
--quiet option 170
quoted numeric string 39
-Qy 18

R

-R 13
-r 10, 13
-r option 169, 175
ranlib 195
Ratfor programs 176
rc, replace text command 253
RCS 178, 182
--rcs option 254
RCS or SCCS files 91
RCS output format 254
RCS rules, terminal 178
rd, delete text command 253
realclean 161
recompilation 83
recompilation of dependent files 164
recompilation, avoiding 164
recompiling 80
--recon option 162, 169
recursion 224
recursive commands 198
recursive expansion 129
recursive invocations of make 167
recursive make 123, 170
recursive make commands 170
recursive make invocations 199
recursive use of make 119, 145
recursively expanded variables 134, 135
Red Hat, contacting iii
redefinition 156
redistribution of input sections 20
redundant context 248
reference 22
references, binding 16
region 46

regular expression suppression 240
regular expressions, matching comparisons 250
-relax 18, 63, 64
relaxing address modes 64
relinking 82
--relocatable 13
relocatable output file 10
relocation 70
relocation level 70
relocations 19, 20
remaking target files 173
replace text lines 245
-retain-symbols-file 19
Revision Control System 254
RFLAGS variable 181
rm utilities 211
RM variable 180
rm, errors 82
-rpath 19
-rpath-link 19
-rpath-link option 19
rule 81
rule chaining 198
rule syntax 94
rule without dependencies or commands 103
rules 88
rules for make 80
runtime linker 19
run-yacc 124

S

-S 14, 19
S 72
-s 14, 19
-S option 170
-s option 170
saving a changed file 275
sbindir 213
SCCS file 201
SCCS files 178, 200
SCCS or RCS files 91
--script= 14

- sdiff options 312
- search directories 8
- search paths 97
- search, implicit rule 175
- SEARCH_DIR 12
- SECT 73
- section attributes 45
- section contents 30
- section fill 47
- section headings 249
- section-alignment 26
- SECTIONS 8, 48
- sections 69
- sections differences 249
- selecting unmerged changes 272
- semicolon 114, 175
- semicolons 31
- serial execution 116
- sh script 207
- shar 162, 228
- shared 20, 24
- shared libraries 20
- shared library 12
- shared objects, locating 19
- sharedstatedir 214
- SHELL 115, 120, 139
- shell 222
- shell commands 115
- shell commands, failed 165
- shell file 89
- shell function 156
- shell metacharacters 256
- SHELL variable 206
- SHELL variables 224
- show-c-function option 250
- show-function-line option 250
- show-function-line=regex option 250
- side by side comparison of files 251
- side by side format 252
- side-by-side option 252
- silent operation 170
- silent option 170
- SIMPLE_BACKUP_SUFFIX environment variable 304
- simply expanded variables 130, 135
- simultaneous commands 116
- single-suffix 189
- single-suffix rule 189
- soname= 11
- sort 221
- sort-common 20
- sorting 20
- source file 80, 175
- source files 89
- space character 249
- space or tab characters 239
- spaces 89
- special built-in target names 104
- special prefix characters 124
- special target 175
- specifying a goal 161
- specifying output file names 13
- speed-large-files option 266
- split-by-file 20
- split-by-reloc 20
- srcdir 216
- S-records 43
- stack 26
- start address 73
- start-group 15
- start-group archives--end-group 15
- static 15
- static pattern rule 106, 185
- static pattern rule, syntax 107
- static pattern rules 107, 200
- static variable 31
- static variables 70
- statistics 20
- stats 20
- stem 185, 187
- stem of a target name 107
- stop option 170
- strings 64
- strip 221
- sub-make 120
- sub-make options 122

- subroutine libraries for linking 193
- subroutines 64
- subst 148, 221
- subst function 106
- substitution references 131
- substitution variable reference 111
- subsystem 26
- suffix 221
- suffix list 175
- suffix rules 185, 189, 198
- SUFFIXES variables 224
- summary output format 240
- SunOS 19, 20
- suppress-common-lines option 252
- SVR4 212
- SVR4 compatibility 18
- switching formats 9
- symbol
 - line number records 70
- symbol information 14, 70
- symbol names 16, 70, 195
- symbol pointer 70
- symbol references 5
- symbol string table, duplications 20
- symbol table 31
- symbol tables
 - processing 68
- symbol tables, caching 17
- symbols 16, 25, 26, 31, 69
 - local 15
 - warnings 21
- symbols, gaps 20
- syntax error 121
- synthesizing instructions 64
- sysconfdir 213
- System V 198
- System V make 201

T

- T 14
- t 14
- t option 162, 170, 259

- tab and blank difference suppression 239
- tabs, unallowed 89
- tabstop alignment 263
- TAGS 162
- tags 209
- TAGS atarget 209
- TANGLE variable 180
- tar 162
- TARGET 10
- target empty commands 175
- target file, modified 170
- target pattern 108, 174, 185, 188
- targets, required 207
- Tbss 21
- Tdata 21
- temp 136
- terminal match-anything pattern rule 189
- terminal rules 184
- test 162
- testing compilation 165
- TEX 178
- TEX and Web 178
- TEX variable 180
- TEXI2DVI 210
- TEXI2DVI variable 180
- Texinfo 178, 210
- Texinfo and Info 178
- Texinfo regular expressions 250
- text 32
- text and binary file comparison 241
- text and data sections, setting 13
- text files, comparing 241
- text manipulation functions 219
- text option 242
- then- part of if-then-else format 257
- time-stamped output 264
- to-file 247
- touch command 170
- touch option 162, 170
- trace-symbol= 15
- traditional formats 20
- traditional-format 20
- transitive closure 198

-Ttext 21
 two column output comparison 252
 two-suffix rule 189
 type descriptor 70
 type information 70

U

-u 14
 Ultrix 212
 --unchanged-group-format= 256
 undefined 155
 undefined symbols 14
 --undefined= 14
 unexport 220
 unified format 246
 unified output format 248
 uninitialized data 31
 uninstall target 208
 unmatched braces 148
 unmatched parentheses 148
 unmerged changes, selecting 272
 updates 246
 updating MAKEFILES 91
 updating software 287
 uppercase usage in variable names 128
 -Ur 13, 14

V

-V 14, 27
 -v 14
 -v option 170
 variable assignments 199
 variable definitions 88
 variable names 128
 variable names, computing 132
 variable reference 148
 variable references 132, 200
 variable values 120
 variable, undefined 171
 variable, using 127
 variable's value 127
 variables 83, 88, 89, 219

variables and functions 128
 variables for overriding commands 211
 variables for overriding options 211
 variables used in implicit rules 179
 variables used in implicit rules, classes 179
 variables, adding more text to definitions 135
 variables, setting 135
 variables, specifying 134
 variables, values 134
 verbatim variable definitions 199
 --verbose 12, 21, 27
 VERSION 61
 -version 14
 version of make, printing 170
 --version option 170, 199
 VERSION_CONTROL environment variable 305
 --version-script= 21
 virtual memory address 30
 VMA 30, 39
 VPATH 98, 151, 179, 197, 206
 vpath 220
 vpath directive 98
 vpath search 200
 VPATH variable 98
 VPATH variables 224
 VxWorks 19

W

-W columns option 252
 -W file option 170
 -W option 163
 -w option 170
 -warn-common 21, 22
 -warn-constructors 22
 warning message 171
 warning messages 90
 warnings 21
 --warn-multiple-gp 22
 -warn-once 23
 --warn-undefined-variables 123
 --warn-undefined-variables option 171
 warranty 170

WEAVE variable 180
Web support site iii
--what-if= option 163
--what-if=file option 170
which are multiples of this number. This defaults to
 512. --heap 25
white space characters 239
white space markers 251
whitespace 31, 89
whitespace, using 125
--whole-archive 23
--width=columns option 252
wildcard 222
wildcard characters 94
wildcard expansion 95
wildcard function 97
wildcard patterns 40
wildcard pitfalls 96
wildcards 194
-W1 9
word 222
words 222
--wrap 23
write protected text bit set 69
writing a semicolon 175

X

-X 15
-x 14

Y

-Y 15
-y 15
-y option 252
Yacc 83, 124, 181
Yacc for C programs 177
YACC variable 180
YACCR variable 180
YFLAGS variable 181

Z

-z initfirst 24
-z interpose 24
-z lodfltr 24
-z nodefaultlib 24
-z nodelete 24
-z nodlopen 24
-z nodump 24
-z now 24
-z origin 24
ZMAGIC 69