Sega Dreamcast

Dreamcast GNUPro™ Toolkit Utilities

Using AS

Using LD

The GNU Binary Utilities

GNU Make

Important Information

This documentation has been provided courtesy of CYGNUS. The contents are applicable to GNUPro™ Toolkit development, however, all references to development support offered by CYGNUS should be ignored.

Technical support for this product as it applies to the Sega Dreamcast™ development environment should be directed to Sega Third Party Developer Technical Support at 415/701-4060. Future updates and/or additional information may also be found at Sega's DTS Website at, **`http//:www.dts.sega.com/NextGen`**

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

Part #: 300-400-1010045

GNUPro warranty

The GNUPro Toolkit is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. This version of GNUPro Toolkit is supported for customers of Cygnus.

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

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

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

How to contact Cygnus

Use the following means to contact Cygnus.

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

Cygnus United Kingdom
36 Cambridge Place
Cambridge CB2 1NS
United Kingdom
Telephone: +44 1223 728728
FAX: +44 1223 728728
email: `info@cygnus.co.uk/`

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

Use the hotline (+1 408 542 9601) to get help, although the most reliable way to resolve problems with GNUPro Toolkit is by using email:

`bugs@cygnus.com`.

Contents

GNUPro Utilities

GNUPro Utilities

Using LD

The GNU Binary Utilities

GNU Make

GNUPro™ Toolkit

Using AS

July, 1998
98r1

CYGNUS

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

1

Overview of as, the GNU assembler

The following documentation serves as a user guide to the GNU assembler, as. It describes what you need to know to use GNU as, covering the syntax expected in source files, including symbols, constants, expressions, and the general directives. This documentation is not intended as an introduction to programming in assembly language—let alone programming in general.

- "Invoking as, the GNU assembler" on page 5
- "Command-line options" on page 13
- "Syntax" on page 31
- "Sections and relocation" on page 41
- "Symbols for the GNU assembler" on page 51
- "Expressions" on page 59
- "Assembler directives" on page 65
- "Machine dependent features" on page 87
- "Acknowledgments" on page 183

The following documentation points to some of the machine-dependent features of the assembler.

- "AMD 29K dependent features" on page 89
- "ARC dependent features" on page 95

This is not an attempt to introduce each machine’s architecture, and neither is it a description of the instruction set, standard mnemonics, registers or addressing modes that are standard to the specific architecture. Consult the manufacturer’s machine architecture documentation for this information.

2

Invoking `as`, the GNU assembler

The GNU Assembler, `as`, is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax.

See the following documentation for more discussion of the GNU assembler.

- "Summary of invocation of as" on page 6
- "Object file formats" on page 7
- "Command line" on page 8
- "Input files" on page 9
- "Output (object) file" on page 10
- "Error and warning messages" on page 11

`as` is primarily intended to assemble the output of the GNU C compiler, `gcc`, for use by the GNU linker, `ld`. Nevertheless, we've tried to make `as` assemble correctly everything that other assemblers for the same machine would assemble.

Any exceptions are documented explicitly (for specific processors and their families, see "Machine dependent features" on page 87). This doesn't mean `as` always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of Motorola 680*x*0 assembly language syntax.

Summary of invocation of as

The following example summarizes how to invoke as. For further details, see “Command-line options” on page 13.

```
as [ -a[cdhlns][=file] ] [ -D ] [ --defsym sym=value ]
   [ -f ] [ --gstabs ][ --help ] [ -I dir ][-J][-K ][-L ]
   [ -o objfile ] [ -R ] [ --statistics ] [ -v ] [ -version ]
   [ --version ] [ -W ] [ -w ] [ -x ] [ -Z ]
   [ -mbig-endian | -mlittle-endian ]
   [ -m[arm]1 | -m[arm]2 | -m[arm]250 | -m[arm]3 | -m[arm]6
                                      | -m[arm]7[t][d]m[I]] ]
   [ -m[arm]v2 | -m[arm]v2a | -m[arm]v3 | -m[arm]v3m | -m[arm]v4
                                                | -m[arm]v4t ]
   [ -mthumb | -mall ]
   [ -mfpa10 | -mfpa11 | -mfpe-old \ -mno-fpu
   [ -EB | -EL ]
   [ -mapcs-32 | -mapcs-26]
   [ -O ]
   [ -Av6 | -Av7 | -Av8 | -Asparclite | -Av9 | -Av9a ]
   [ -xarch=v8plus | -xarch=v8plusa ] [ -bump ]
   [ -ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC ]
   [ -b ] [ -no-relax ]
   [ -l ] [ -m68000 | -m68010 | -m68020 |...]
   [ -nocpp ] [ -EL ] [ -EB ] [ -G num ] [ -mcpu=CPU ]
   [ -mips1 ] [ -mips2 ] [ -mips3 ] [ -m4650 ] [ -no-m4650 ]
   [ --trap ] [ --break ]
   [ --emulation= name ]
   [ -- | files...]
```

Unlike older assemblers, as is designed to assemble a source program in one pass of the source file. This has a subtle impact on the .org directive (see “.org new-lc, fill” on page 78).

Object file formats

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See “Symbol attributes” on page 57.

Command line

After the program name as, the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

`--` (two hyphens), by themselves, name the standard input file explicitly, as one of the files for `as` to assemble. Except for `--`, any command line argument that begins with a hyphen (`-`) is an option. Each option changes the behavior of `as`. No option changes the way another option works. An option is a `-` followed by one or more letters; the *case* of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). The following two command lines are equivalent.

```
as -o my-object-file.o mumble.s
as -omy-object-file.o mumble.s
```

Input files

We use the phrase *source program*, abbreviated *source*, to describe the program input to one run of `as`. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified. Each time you run `as`, it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give `as` a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give as no file names it attempts to read one input file from the as standard input, which is normally your terminal. You may have to type Ctrl-D to tell as there is no more program to assemble.

Use '`--`' if you need to explicitly name the standard input file in your command line.

If the source is empty, as produces a small, empty object file.

Filenames and line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See "Error and warning messages" on page 11.

Physical files are those files named in the command line given to as.

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when `as` source is itself synthesized from other files. See ".app-file string" on page 68.

Output (object) file

Every time you run `as`, it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is `a.out`, or `b.out`, when `as` is configured for the Intel 80960. You can give it another name by using the `-o` option. Conventionally, object file names end with `.o`. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn’t currently possible, but it can be done for the `a.out` format.)

The object file is meant for input to the linker `ld`. It contains assembled program code, information to help `ld` integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

Error and warning messages

`as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs as automatically. Warnings report an assumption made so that `as` could keep assembling a flawed program; errors report a grave problem that stops the assembly. Warning messages have the following format (where *NNN* is a line number).

```
file_name:NNN:Warning Message Text
```

If a logical file name has been given (see ".app-file string" on page 68) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see ".line line-number" on page 75) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the Unix tradition).

Error messages have the following format.

```
file_name:NNN:FATAL:Error Message Text
```

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

3

Command-line options

The following documentation describes command-line options available in all versions of the GNU assembler; see "Summary of invocation of `as`" on page 6 for a complete invocation declaration example.

See also “Machine dependent features” on page 87 for a list of machines in order to find documentation specific to each architecture’s particular options.

If you are invoking `as` using the GNU C compiler (version 2), you can use the `-Wa` option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the `-Wa`) by commas. Use the following for example.

```
gcc -c -g -O -Wa,-alh,-L file.c
```

This input emits a listing to standard output with high-level and assembly source. Usually you do not need to use this `-Wa` mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the `-v` option to see precisely what options it passes to each compilation pass, including the assembler.)

Enable listings: -a[cdhlns]

These options enable listing output from the assembler. By itself, -a requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: -ah requests a high-level language listing, -al requests an output-program assembly listing, and -as requests a symbol table listing. High-level listings require that a compiler debugging option like -g be used, and that also assembly listings (-al) be requested. Use the -ad option to omit debugging directives from the listing.

Use the –ac option to omit false conditionals from a listing. Any lines which are not assembled because of a false .if (or .ifdef, or any other conditional), or a true .if followed by a .else, will be omitted from the listing.

Use the –ad option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives .list, .nolist, .psize, .eject, .title, and .sbttl. The -an option turns off all forms processing. If you do not request listing output with one of the -a options, the listing-control directives have no effect. The letters after -a may be combined into one option, such as -aln.

-D

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with `as`.

Work faster: `-f`

`-f` should only be used when assembling programs written by a (trusted) compiler. `-f` stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. See “Preprocessing” on page 32.

WARNING: If you use `-f` when the files actually need to be pre-processed (if they contain comments, for example), as does not work correctly.

`.include` search path: `-I` *`path`*

Use this option to add a *`path`* to the list of directories as searches for files specified in `.include` directives (see "`.include "file"`" on page 74). You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, as searches any `-I` directories in the same order as they were specified (left to right) on the command line.

Difference tables: `-K`

as sometimes alters the code emitted for directives of the form `.word` *`sym1-sym2`*; see “`.word expressions`” on page 86. You can use the `-K` option if you want a warning issued when this is done.

Include local labels: -L

Labels beginning with `L` (upper case only) are called *local labels*. See “Symbol names” on page 54. Normally, you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `as` and `ld` discard such labels, so you do not normally debug with them.

This option tells `as` to retain those `L...` symbols in the object file. Usually if you do this you also tell the linker, `ld`, to preserve symbols whose names begin with `L`. By default, a local label is any label beginning with `L`, but each target is allowed to redefine the local label prefix. On the HPPA, local labels begin with `L$`; `;` for the ARM family.

Assemble in MRI compatibility mode: `-M`

The `-M` or `--mri` option selects Microtec Research Inc.'s (MRI's) compatibility mode. This changes the syntax and pseudo-op handling of `as` to make it compatible with the `ASM68K` or the `ASM960` (depending upon the configured target) assembler from MRI.

The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information.

The purpose of this option is to permit assembling existing MRI assembler code using `as`.

The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be sup-ported using other object file formats.

Supporting these would require enhancing each object file format individually. These are the following.

- *global symbols in common section*
  The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. as handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them.
- *complex relocations*
  The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not support by other object file formats.
- `END` *pseudo-op specifying start address*
  The MRI `END` pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the `-e` option to the linker, or in a linker script.
- `IDNT`, `.ident` and `NAME` *pseudo-ops*
  The MRI `IDNT`, `.ident` and `NAME` pseudo-ops assign a module name to the output file. This is not supported by other object file formats.
- `ORG` *pseudo-op*
  The m68k MRI `ORG` pseudo-op begins an absolute section at a given address. This differs from the usual `as .org` pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

There are some other features of the MRI assembler which are not supported by `as`, typically either because they are difficult or because they seem of little consequence. Some of these may be supported in future releases.

- *EBCDIC strings*
  EBCDIC strings are not supported.
- *Packed binary coded decimal*
  Packed binary coded decimal is not supported. This means that the `DC.P` and `DCB.P` pseudo-ops are not supported.
- `FEQU` *pseudo-op*
  The m68k `FEQU` pseudo-op is not supported.
- `NOOBJ` pseudo-op
  The m68k `NOOBJ` pseudo-op is not supported.
- `OPT` branch control options
  The m68k `OPT` branch control options—`B`, `BRS`, `BRB`, `BRL`, and `BRW`– are ignored. `as` automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose.
- `OPT` list control options
  The following m68k OPT list control options are ignored: `C`, `CEX`, `CL`, `CRE`, `E`, `G`, `I`, `M`, `MEX`, `MC`, `MD`, `X`.
- Other `OPT` options
  The following m68k `OPT` options are ignored: `NEST`, `O`, `OLD`, `OP`, `P`, `PCO`, `PCR`, `PCS`, `R`.
- `OPT D` option is default
  The m68k `OPT D` option is the default, unlike the MRI assembler. `OPT NOD` may be used to turn it off.
- `XREF` pseudo-op.
  The m68k `XREF` pseudo-op is ignored.
- `.debug` pseudo-op
  The i960 `.debug` pseudo-op is not supported.
- `.extended` pseudo-op
  The i960 `.extended` pseudo-op is not supported.
- `.list` pseudo-op.
  The various options of the i960 `.list` pseudo-op are not supported.
- `.optimize` pseudo-op
  The i960 `.optimize` pseudo-op is not supported.
- `.output` pseudo-op
  The i960 `.output` pseudo-op is not supported.

- `.setreal` pseudo-op
  The i960 `.setreal` pseudo-op is not supported.

Dependency tracking: `--MD`

`as` can generate a depenedency file for the file that it creates. This file consists of a single rule suitable for **`make`**, describing the dependencies for the main source file.

The rule is written to the file named in its argument.

This feature is used in the automatic updating of files.

Name the object file: -o

There is always one object file output when you run `as`. By default it has the name `a.out` (or `b.out`, for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, `as` overwrites any existing file of the same name.

Join data and text sections: `-R`

`-R` tells `as` to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See “Sections and relocation” on page 41.) When you specify `-R`, it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of `as`. In future, `-R` may work this way.

When `as` is configured for COFF output, this option is only useful if you use sections named `.text` and `.data`.

`-R` is not supported for any of the HPPA targets. Using `-R` generates a warning from `as`.

Display sssembly statistics: `--statistics`

Use `--statistics` to display two statistics about the resources used by `as`: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

Announce version: -v

You can find out what version of as is running by including the -v option (which you can also spell as -version) on the command line.

Suppress warnings: `-W`

`as` should never give a warning or error message when assembling compiler output. However, programs written by people often cause `as` to give a warning that a particular assumption was made. All such warnings are directed to the standard error file. If you use this option, `-W`, no wrnings are issuedd. This option only affects the warning messages and it does not change any particular of how `as` assembles your file. Errors that stop the assembly are still reported.

Generate object file in spite of errors: -Z

After an error message, as normally produces no output. If for some reason you are interested in object file output even after as gives an error message on your program, use the -Z option. If there are any errors, as continues, and writes an object file after a final warning message of the following form.

```
n errors, m warnings, generating bad object file.
```

4

Syntax

The following documentation describes the machine-independent syntax allowed in a source file.

`as` syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler, except that `as` does not assemble Vax bit-fields.

Preprocessing

The `as` internal preprocessor has the following functionality.

- Adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- Removes all comments, replacing them with a single space, or an appropriate number of newlines.
- Converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (see ".include "file"" on page 74). You can use the GNU C compiler driver to get other `cpp`-style preprocessing by giving the input file a `.S` suffix. See "Options Controlling the Kind of Output" in *Using GNU CC* in ***GNUPro Compiler Tools***.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is `#NO_APP` or if you use the `-f` option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the by putting a line that says `#APP` before the text that may contain whitespace or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intend to support `asm` statements in compilers whose output is otherwise free of comments and whitespace.

Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see "Character constants" on page 36), any whitespace means the same as exactly one space.

Comments

There are two ways of rendering comments to `as`. In both cases the comment is equivalent to one space.

Anything from `/*` through the next `*/` is a comment, as in the following statement.

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/
```

This means you may not nest the following types of comments.

```
/* This sort of comment does not nest. */
```

Anything from the *line comment* character to the next newline is considered a comment and is ignored.

The line comment character is '`#`' for the Vax; '`#`' for the i960; '`!`' for the SPARC family; '`|`' for the Motorola 68K family; '`;`' for the AMD 29K family; '`;`' for the H8/300 family; '`!`' for the H8/500 family; '`;`' for the HPPA family; '`!`' for the Hitachi SH family; '`!`' for the Z8000 family, and '`;`' for the ARC family; see "Machine dependent features" on page 87 to locate specific families.

On some machines there are two different line comment characters. One character only begins a comment if it is the first non-whitespace character on a line, while the other always begins a comment.

To be compatible with past assemblers, lines that begin with '`#`' have a special interpretation. Following the '`#`' should be an absolute expression (see "Expressions" on page 59): the logical line number of the *next* line. Then a string (see "Strings" on page 36) is allowed: if present, it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                  # This is an ordinary comment.
# 42-6 "new_file_name"# New logical file name
                  # This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of `as`.

Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters '`_.$`'. On most machines, you can also use `$` in symbol names; exceptions are noted for each architecture; see "Machine dependent features" on page 87 for specific processor families.

WARNING: No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant.

Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See "Symbols for the GNU assembler" on page 51.

Statements

A statement ends at a newline character (`\n`) or line separator character. (The line separator is usually (`;`), unless this conflicts with the comment character. Exceptions are noted for each architecture; see "Machine dependent features" on page 87.) The newline or separator character is considered part of its preceding statement.

NOTE: Newlines and separators within character constants are an exception: *they do not end statements*.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash (`\`) immediately in front of any newlines within the statement. When `as` reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot (`.`), then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language instruction: it assembles into a machine language *instruction*. Different versions of as for different computers recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is a symbol immediately followed by a colon (`:`). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. See "Labels" on page 52.

For HPPA targets, labels need not be immediately followed by a colon, but the definition of a label must begin in column zero, or the beginning of the line. This also implies that only one label may be defined on each line. Use the following statement as an example.

```
label:       .directivefollowed by something
another_label:        # This is an empty statement.
                        instruction    operand_1, operand_2, ...
```

Constants

A constant is a number, written so that its value is known by inspection, without knowing any context, like the following input example.

```
.byte   74, 0112, 092, 0x4A, 0X4a,'J, '\J  # All the same value.
.ascii  "Ring the bell\7"                  # A string constant.
.octa   0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float  0f-314159265358979323846264338327\
95028841971.693993751E-40                  # - pi, a flonum.
```

Character constants

There are two kinds of character constants. A character stands for one character in one byte and its value may be used in *numeric expressions*. *String constants* (properly called *string literals*) are potentially many bytes and their values may not be used in *arithmetic expressions*.

Strings

A string is written between double-quotes. It may contain double-quotes or null characters.

The way to get special characters into a string is to escape these characters: precede them with a backslash (`\`) character. For example `\\` represents one backslash: the first `\` is an escape which tells as to interpret the second character literally as a backslash (which prevents as from recognizing the second `\` as an escape character).

The complete list of escapes follows.

`\b`
: Mnemonic for backspace; for ASCII this is octal code 010.

`\f`
: Mnemonic for FormFeed; for ASCII this is octal code 014.

`\n`
: Mnemonic for newline; for ASCII this is octal code 012.

`\r`
: Mnemonic for carriage-Return; for ASCII this is octal code 015.

`\t`
: Mnemonic for horizontal Tab; for ASCII this is octal code 011.

`\` *`digit digit digit`*
: An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, `\008` has the value 010, and `\009` the value 011.

`\x` *`hex-digit hex-digit`*

A hex character code. The numeric code is 2 hexadecimal digits. Either upper or lower case `x` works.

`\\`
: Represents one `\` character.

`\"`
: Represents one `"` character. Needed in strings to represent this character, because an unescaped `"` would end the string.

`\ anything-else`
: Any other character when escaped by `\` gives a warning, but assembles as if the `\` was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However `as` has no other interpretation, so `as` knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

Characters

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write `'\\` where the first `\` escapes the second `\`. As you can see, the quote is an 'acute' accent, not a 'grave' accent (`` ` ``). A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. `as` assumes your character code is ASCII: `'A` means 65, `'B` means 66, and so on.

Number constants

`as` distinguishes three kinds of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an int in the C language. *Bignums* are integers, but they are stored in more than 32 bits. *Flonums* are floating point numbers; see "Flonums" on page 38.

Integers

A binary integer is `0b` or `0B` followed by one or more of the binary digits, `01`.

`0b`	invalid
`0b0`	valid

An octal integer is 0 followed by zero or more of the octal digits (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`).

A decimal integer starts with a non-zero digit followed by zero or more digits (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`).

A hexadecimal integer is `0x`, or `0X`, followed by zero or more hexadecimal digits chosen from `0123456789abcdefABCDEF`. Integers have the usual values. To denote a negative integer, use the prefix operator (`-`), discussed under expressions (see “Prefix operators” on page 63).

`0x`	valid=`0x0`
`0x1`	valid

Bignums

A *bignum* has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

Flonums

A *flonum* represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by `as` to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer’s floating point format (or formats) by a portion of `as` specialized to that computer.

A flonum is written by writing (in order) the following:

- The digit `0`. (`0` is optional on the HPPA.)
- A letter, to tell `as` the rest of the number is a flonum. `e` is recommended. Case is not important.
  - On the H8/300, H8/500, Hitachi SH, and AMD 29K architectures, the letter must be one of the letters: `D`, `F`, `P`, `R`, `S`, or `X` (in uppercase or lower case).
  - On the ARC, the letter must be one of the letters: `D`, `F`, `R`, or `S` (in uppercase or lowercase).
  - On the Intel 960 architecture, the letter must be one of the letters: `D`, `F`, or `T` (in uppercase or lower case).
  - On the HPPA architecture, the letter must be `E` (uppercase only).
- An optional sign: either `+` or `-`.
- An optional integer part: zero or more decimal digits.
- An optional fractional part: (`.`) followed by zero or more decimal digits.
- An optional exponent, consisting of the following elements:

- An `E` or `e`.
- Optional sign: either `+` or `-`.
- One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

`as` does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running `as`.

5

Sections and relocation

A *section* is a range of addresses, with no gaps; all data in those addresses is treated the same for some particular purpose (for example, there may be a *read-only* section).

The following documentation discusses sections and their relocation.

The linker, `ld`, reads many object files (partial programs) and combines their contents to form a runnable program. When `as` emits an object file, the partial program is assumed to start at address, `0`. `ld` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification of relocation, but it suffices to explain how `as` uses sections.

`ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. For the H8/300 and H8/500, and for the Hitachi SH, **`as`** pads sections if needed to ensure they end on a word (sixteen bit) boundary.

An object file written by `as` has at least three sections, any of which may be empty. These are named *text*, *data* and *bss* sections.

When it generates COFF output, `as` can also generate whatever other named sections you specify using the `.section` directive (see “.section name” on page 80). If you do not use any directives that place output in the `.text` or `.data` sections, these sections still exist, but are empty.

When `as` generates SOM or ELF output for the HPPA, as can also generate whatever other named sections you specify using the `.space` and `.subspace` directives.

See ***HP9000 Series 800 Assembly Language Reference Manual*** (HP 92432-90001) for details on the `.space` and `.subspace` assembler directives.

Additionally, as uses different names for the standard text, data, and bss sections when generating SOM output. Program text is placed into the `$CODE$` section, data into `$DATA$`, and BSS into `$BSS$`.

Within the object file, the text section starts at address `0`, the data section follows, and the bss section follows the data section.

When generating either SOM or ELF output files on the HPPA, the text section starts at address `0`, the data section at address `0x4000000`, and the bss section follows the data section.

To let `ld` know which data changes when the sections are relocated, and how to change that data, `as` also writes to the object file details of the relocation needed. To perform relocation, each time an address in the object file is mentioned, `ld` must know the following criteria.

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of (*`address`*)-(*`start-address of section`*)?
- Is the reference to an address “Program-Counter relative”?

  In fact, every address `as` ever uses is expressed as the following.

  ```
  (section) + (offset into section).
  ```

Further, most expressions `as` computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.) In this manual we use the notation `{`*`secname N`*`}` to mean “offset *`N`* into section, *`secname`*.”

Apart from text, data and bss sections you need to know about the *absolute* section. When `ld` mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is “relocated” to run-time address 0 by `ld`. Although the linker never arranges two partial programs’ data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the program.

The idea of sections is extended to the *undefined* section. Any address whose section is unknown at assembly time is by definition rendered {undefined *U*}– where *U* is filled in later.

Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy the word *section* is used to describe groups of sections in the linked program. `ld` puts all partial programs’ text sections in contiguous addresses in the linked program.

It is customary to refer to the text section of a program, meaning all the addresses of all partial programs’ *text sections*. Likewise for data and bss sections.

Some sections are manipulated by `ld`; others are invented for use by the assembler and have no meaning except during assembly.

`ld` sections

`ld` deals with just four kinds of sections, summarized by the following .

named sections
text section
data section

These sections hold your program. as and ld treat them as separate but equal sections. Anything you can say of one section is true for another. When the program is running, however, it is customary for the text section to be unalterable. The text section is often shared among processes: it contains instructions, constants and the like. The data section of a running program is usually alterable; for example, C variables would be stored in the data section.

bss section

This section contains zeroed bytes when your program begins running. It is used to hold uninitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

absolute section

Address 0 of this section is always *relocated* to runtime address 0. This is useful if you want to refer to an address that `ld` must not change when relocating. In this sense we speak of absolute addresses being *unrelocatable*: they do not change during relocation.

undefined section

This *section* is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. The following example uses the traditional section names `.text` and `.data`. Memory addresses are on the horizontal axis.

Partial program #1

```
 text                   data                  bss
|ttttt                 |dddd                 |00                |
```

Partial program #2

```
 text                   data                  bss
|TTT                   |DDDD                 |000               |
```

Linked program

```
    text                data                  bss
|  | TTT   | ttttt   |  |dddd      | DDDD     |00000 ...          |
```

Addresses: 0...

as internal sections

These sections are meant only for the internal use of `as`. They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in `as` warning messages, so it might be helpful to have an idea of their meanings to `as`. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

ASSEMBLER-INTERNAL-LOGIC-ERROR!
: An internal assembler logic error has been found. This means there is a bug in the assembler.

expr *section*
: The assembler stores complex expression internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the `expr` section.

Sub-sections

Assembled bytes conventionally fall into two sections: text and data. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. `as` allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0` before each section of code being output, and a `.text 1` before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes. (Sub-sections may be padded a different amount on different flavors of `as`.)

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; `ld` and other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text` *`expression`* or a `.data` *`expression`* statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: `.section` *`name`*, *`expression`*. *`expression`* should be an absolute expression. (See "Expressions" on page 59.) If you just say `.text`, then `.text 0` is assumed. Likewise, `.data` means `.data 0`. Assembly begins in `text 0`. For instance, use the following example.

```
.text    0     # The default subsection is text 0 anyway.
.ascii         "This lives in the first text subsection. *"
.text    1
.ascii         "But this lives in the second text subsection."
.data    0
.ascii         "This lives in the data section,"
.ascii         "in the first data subsection."
.text    0
.ascii         "This lives in the first text section,"
.ascii         "immediately following the asterisk (*)."
```

Each section has a *location counter* incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to `as`, there is no concept of a subsection location counter. There is no way to directly manipulate a location counter—but the `.align` directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the *active* location counter.

bss section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes. Addresses in the bss section are allocated with special directives; you may not assemble anything directly into the bss section. Hence there are no bss subsections.

See “.comm symbol, length” on page 69.

6

Symbols for the GNU assembler

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug. The following documentation discusses more about symbols and the assembler.

- "Labels" on page 52
- "Giving symbols other values" on page 53
- "Symbol names" on page 54
- "The special dot symbol" on page 56
- "Symbol attributes" on page 57

WARNING: `as` does not place symbols in the object file in the same order they were declared. This may break some debuggers.

Labels

A *label* is written as a symbol immediately followed by a colon (`:`). The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

On the HPPA, the usual form for a label need not be immediately followed by a colon, but instead must start in column zero. Only one label may be defined on a single line. To work around this, the HPPA version of `as` also provides a special directive, `.label`, for defining labels more flexibly.

Giving symbols other values

A symbol can be given an arbitrary value by writing a symbol, followed by an equal sign (=), followed by an expression (see “Expressions” on page 59). This is equivalent to using the `.set` directive. See “.set symbol, expression” on page 82.

Symbol names

Symbol names begin with a letter or with either a dot (`.`) or an underscore (`_`); on most machines, you can also use **`$`** in symbol names; exceptions are noted for each architecture in the documentation; see “Machine dependent features” on page 87 to locate a specific architecture. That character may be followed by any string of digits, letters, dollar signs (unless exceptions are noted for each architecture), and underscores. For the AMD 29K family, **`?`** is also allowed in the body of a symbol name, though not at its beginning. Case of letters is significant: **`foo`** is a different symbol name than **`Foo`**.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

Local symbol names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names ‘**`0`**’ ‘**`1`**’`...`‘**`9`**’. To define a local symbol, write a label of the form ‘*`N`*`:`’ (where *`N`* represents any digit). To refer to the most recent previous definition of that symbol write ‘*`N`***`b`**’, using the same digit as when you defined the label. To refer to the next definition of a local label, write ‘*`N`***`f`**’—where *`N`* gives you a choice of 10 forward references. The `b` stands for *backwards* and the **`f`** stands for *forwards*.

Local symbols are not emitted by the current GNU C compiler.

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have the following parts.

`L`
: All local labels begin with `L`. Normally both `as` and `ld` forget symbols that start with `L`. These labels are used for symbols you are never intended to see. If you use the `-L` option then `as` retains these symbols in the object file. If you also instruct `ld` to retain these symbols, you may use them in debugging.

`digit`
: If the label is written `0:` then the digit is `0`. If the label is written `1:` then the digit is `1`. And so on up through `9`:.

`^A`
: This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value `\001`.

ordinal number
: This is a serial number to keep the labels distinct. The first `0:` gets the number `1`; The 15th `0:` gets the number `15`; etc.. Likewise for the other labels `1:` through `9:`.

For instance, the first `1:` is named `L1^A1`, the 44th `3:` is named `L3^A44`.

The special dot symbol

The special symbol "`.`" refers to the current address into which `as` is assembling. Thus, the expression, `melvin: .long`, defines `melvin` to contain its own address.

Assigning a value to "`.`" is treated the same as a `.org` directive. Thus, the expression of '**`.=.+4`**' is the same as saying '`.space 4`'.

Symbol attributes

Every symbol has, as well as its name, the attributes, *Value* and *Type*. Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, `as` assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as `ld` changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and `ld` tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a `.comm` common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

Symbol attributes for `a.out`

The following documentation discusses the symbol attributes for the `a.out` file format.

Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (see ".desc symbol, abs-expression" on page 70). A descriptor value means nothing to `as`.

Other

This is an arbitrary 8-bit value. It means nothing to `as`.

Symbol attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between `.def` and `.endef` directives.

Primary attributes

The symbol name is set with `.def`; the value and type, respectively, with `.val` and `.type`.

Auxiliary attributes

The `as` directives `.dim`, `.line`, `.scl`, `.size`, and `.tag` can generate auxiliary symbol table information for COFF.

Symbol attributes for SOM

The SOM format for the HPPA supports a multitude of symbol at-tributes set with the `.EXPORT` and `.IMPORT` directives. The attributes are described in ***HP9000 Series 800 Assembly Language Reference Manual*** (HP 92432-90001) with the “IMPORT and EXPORT assembler directive” documentation.

7

Expressions

An *expression* specifies an address or numeric value. Whitespace may precede and/or follow an expression. The following documentation describes more about expressions and the assembler.

The result of an expression must be an absolute number, or else an off-set into a particular section. If an expression is not absolute, and there is not enough information when as sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression—but the second pass is currently not implemented. `as` aborts with an error message in this situation.

Empty expressions

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression, and `as` assumes a value of (absolute) 0. This is compatible with other assemblers.

Integer expressions

An *integer expression* is one or more *arguments* delimited by *operators*.

Arguments

Arguments are symbols, numbers or sub-expressions. In other contexts, *arguments* are sometimes called *arithmetic operands*. In this manual, to avoid confusing them with the *instruction operands* of the machine language, we use the term *argument* to refer to parts of expressions only, reserving the word *operand* to refer only to machine instruction operands.

Symbols are evaluated to yield {*section NNN*} where *section* is one of text, data, bss, absolute, or undefined. *NNN* is a signed, 2's complement 32 bit integer. Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and `as` pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Sub-expressions are a left parenthesis '`(`' followed by an integer expression, followed by a right parenthesis '`)`'; or a prefix operator followed by an argument.

Operators

Operators are arithmetic functions, like `+` or `%`. Prefix operators are followed by an argument (see "Prefix operators"). Infix operators appear between their arguments (see "Infix operators"). Operators may be preceded and/or followed by whitespace.

Prefix operators

`as` has the following *prefix operators*. They each take one argument, which must be absolute.

`-` (*Negation*)
: Two's complement negation.

`~` (*Complementation*)
: Bitwise not.

Infix operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from `+` or `-`, both arguments must be absolute, and the result is absolute.

- ***Highest Precedence***

  `*` *(Multiplication)*
  `/` *(Division)*

  Truncation is the same as the C operator '`/`'.|

  `%` *(Remainder)*
  `<`
  `<<` *(Shift Left)*

  Same as the C operator '`<<`'.

  `>`
  `>>` *(Shift Right)*

  Same as the C operator '`>>`'.

- ***Intermediate Precedence***

  `|` *(Bitwise Inclusive Or)*
  `&` *(Bitwise And)*
  `^` *(Bitwise Exclusive Or)*
  `!` *(Bitwise Or Not)*

- ***Lowest Precedence***

  + *(Addition)*

  If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

  - *(Subtraction)*

  If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

  In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

8

Assembler directives

All assembler directives have names that begin with a period (`.`). The rest of the name is letters, usually in lower case.

The following documentation lists and shows the location of directives that are available regardless of the target machine configuration for the GNU assembler. For additional directives pertinent to each architecture, see “Machine dependent features” on page 87.

One day the following directives won’t work. They are included for compatibility with older assemblers.

```
.abort
.app-file
.line
```

See the following documentation for specific information on the previously listed assembler directives.

.abort

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells as to quit also. One day `.abort` will not be supported.

.ABORT

When producing COFF output, `as` accepts this directive as a synonym for `.abort`.

When producing `b.out` output, `as` accepts this directive, but ignores it.

.align *abs-expr, .abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described in the following. The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.

The way the required alignment is specified varies from system to system. For the a29k, hppa, m86k, m88k, w65, sparc, and Hitachi SH, and i386 using ELF format, the first expression is the alignment request in bytes. For example, `.align 8` advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. For other systems, including the i386 using a.out format, it is the number of low-order zero bits the location counter must have after advancement. For example `.align 3` advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides `.balign` and `.p2align` directives, which have a consistent behavior across all architectures (but are specific to GAS).

.app-file string

`.app-file` (which may also be spelled '`.file`') tells as that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '"'; but if you wish to specify an empty file name is permitted, you must give the quotes–" ". This statement may go away in future: it is only recognized to be compatible with old as programs.

.ascii "*string*"...

`.ascii` expects zero or more string literals (see "Strings" on page 36) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

`.asciz` *“string”*...

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The `z` in `.asciz` stands for *zero*.

`.balign[wl]` *abs-expr, abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example `.balign 8` advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

`.byte` *expressions*

`.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

`.comm` *symbol, length*

`.comm` declares a common symbol named *`symbol`*. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If `ld` does not see a definition for the symbol—just one or more common symbols—it will allocate *`length`* bytes of uninitialized memory. *`length`* must be an absolute expression. If `ld` sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF, the .comm directive takes an optional third argument. This is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be 0). The alignment must be an absolute expression, and it must be a power of 2. If `ld` allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, `as` will set the alignment to the largest power of 2 less than or equal to the size of the symbol, up to a maximum of 16.

The syntax for `.comm` differs slightly on the HPPA. The syntax is *`symbol`*`.comm,` *`length`*; symbol is optional.

`.data` *`subsection`*

`.data` tells as to assemble the following statements onto the end of the data subsection numbered *`subsection`* (which is an absolute expression). If *`subsection`* is omitted, it defaults to zero.

`.def` *`name`*

Begin defining debugging information for a symbol *`name`*; the definition extends until the `.endef` directive is encountered.

This directive is only observed when as is configured for COFF format output; when producing `b.out`, `.def` is recognized, but ignored.

`.desc` *`symbol, abs-expression`*

This directive sets the descriptor of the symbol (see“Symbol attributes” on page 57) to the low 16 bits of an absolute expression.

The `.desc` directive is not available when as is configured for COFF output; it is only for `a.out` or `b.out` object format. For the sake of compatibility, `as` accepts it, but produces no output, when configured for COFF.

`.dim`

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. `.dim` is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

.double *flonums*

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how `as` is configured. For each architecture's documentation, see "Machine dependent features" on page 87.

.eject

Force a page break at this point, when generating assembly listings.

.else

`.else` is part of the as support for conditional assembly; see ".if absolute expression" on page 73. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

.endef

This directive flags the end of a symbol definition begun with `.def`.

`.endef` is only meaningful when generating COFF format output; if `as` is configured to generate `b.out`, it accepts this directive but ignores it.

.endif

`.endif` is part of the as support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See ".if absolute expression" on page 73.

.equ *symbol, expression*

This directive sets the value of *`symbol`* to *`expression`*. It is synonymous with `.set`; see ".set symbol, expression" on page 82. The syntax for `equ` on the HPPA is *`symbol.equ expression`*.

.equiv *symbol, expression*

The `.equiv` directive is like the `.equ` and the `.set` directives, except the assembler will signal an error if *`symbol`* is already defined.

Except for the contents of the error message, the following input is roughly equivalent.

```
.ifdef SYM
```

```
.err
.endif
.equ SYM,VAL
```

.err

If `as` assembles a `.err` directive, it will print an error message and, unless the `–Z` option was used, it will not generate an object file. This can be used to signal error an unconditionally compiled code..

.extern

`.extern` is accepted in the source program—for compatibility with other assemblers—but it is ignored. `as` treats all undefined symbols as external.

.file *string*

`.file` (which may also be spelled `.app-file`) tells `as` that we are about to start a new logical file. *`string`* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes (`"`); but if you wish to specify an empty file name, you must give the quotes– `""`. This statement may go away in future; it is only recognized to be compatible with old `as` programs. In some configurations of as, `.file` has already been removed to avoid conflicts with other assemblers. For each architecture's documentation, see "Machine dependent features" on page 87.

.fill *repeat, size, value*

`result`, *`size`* and *`value`* are absolute expressions. This emits *`repeat`* copies of *`size`* bytes. *`repeat`* may be zero or more. *`size`* may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each *`repeat`* bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are *`value`* rendered in the byte-order of an integer on the computer `as` is assembling for. Each *`size`* bytes in a repetition is taken from the lowest order *`size`* bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

`size` and *`value`* are optional. If the second comma and value are absent, *`value`* is assumed zero. If the first comma and following tokens are absent, *`size`* is assumed to be 1.

.float *flonums*

`.float` assembles zero or more flonums, separated by commas. It has the same effect as `.single`. The exact kind of floating point numbers emitted depends on how `as` is configured. For each architecture's documentation, see “Machine dependent features” on page 87.

.global *symbol*, .globl *symbol*

`.global` makes the symbol visible to `ld`. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* takes its attributes from a symbol of the same name from another file linked into the same program. Both spellings (`.globl` and `.global`) are accepted, for compatibility with other assemblers.

On the HPPA, `.global` is not always enough to make it accessible to other partial programs. You may need the HPPA-only `.EXPORT` directive as well. See “HPPA assembler directives” on page 139.

.hword *expressions*

`.hword` expects zero or more expressions, and emits a 16 bit number for each.

This directive is a synonym for `.short`; depending on the target architecture, it may also be a synonym for `.word`.

.ident

`.ident` is used by some assemblers to place tags in object files. `as` simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

.if *absolute expression*

`.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an absolute expression) is non-zero. The end of the conditional section of code must be marked by `.endif` (see “.endif” on page 71); optionally, you may include code for the alternative condition, flagged by `.else` (see “.else” on page 71). The following variants of `.if` are also supported:

`.ifdef` *symbol*
: Assembles the following section of code if the specified *symbol* has been defined.

`.ifndef` *`symbol`*

`ifnotdef` *`symbol`*

Assembles the following section of code if the specified *`symbol`* has not been defined. Both spelling variants are equivalent.

.include “*file*”

`.include` provides a way to include supporting files at specified points in your source program. The code from *`file`* is assembled as if it followed the point of the .include; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I` command-line option (see “.include search path: -I path” on page 18). Quotation marks are required around *`file`*.

.int *expressions*

Expect zero or more *`expressions`*, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly uses.

.irp *symbol*, *values*...

Evaluate a sequence of statements assigning different values to *`symbol`*. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each *`value`*, *`symbol`* is set to *`value`*, and the sequence of statements is assembled.

If no *`value`* is listed, the sequence of statements is assembled once, with *`symbol`* set to the null string. To refer to *`symbol`* within the sequence of statements, use `\`*`symbol`*.

```
.irp        param,1,2,3
move        d\param,sp@-
.endr
```

For example, assembling the previous statement is equivalent to assembling the following.

```
move        d1,sp@-
move        d2,sp@-
move        d3,sp@-
```

.irpc *symbol, values...*

Evaluate a sequence of statements assigning different values to `symbol`. The sequence of statements starts at the .irpc directive, and is terminated by an .endr directive. For each character in `value`, `symbol` is set to the character, and the sequence of statements is assembled. If no `value` is listed, the sequence of statements is assembled once, with `symbol` set to the null string. To refer to `symbol` within the sequence of statements, use `\symbol`.

```
.irpc        param,123
move         d\param,sp@-
.endr
```

For example, assembling the previous statement is equivalent to assembling the following declaration.

```
move         d1,sp@-
move         d2,sp@-
move         d3,sp@-7.32
```

.lcomm *symbol, length*

`.lcomm` Reserve `length` (an absolute expression) bytes for a local common denoted by `symbol`. The section and value of `symbol` are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. `symbol` is not declared global (see “.global symbol, .globl symbol” on page 73), so it’s normally not visible to `ld`.

The syntax for `.lcomm` differs slightly on the HPPA. The syntax is `symbol.lcomm, length`; `symbol` is optional.

.lflags

`as` accepts this directive, for compatibility with other assemblers, but ignores it.

.line *line-number*

Change the logical line number. `line-number` must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number `line-number`-1. One day `as` will no longer support this directive: it is recognized only for compatibility with existing assembler programs.

WARNING: In the AMD29K configuration of as, this command is not available; use the synonym `.ln` in that context.

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, `as` still recognizes it when producing COFF output, and treats `.line` as though it were the COFF `.ln` if it is found outside a `.def`/`.endef` pair. Inside a `.def`, `.line` is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

.list

Control (in conjunction with the `.nolist` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero. By default, listings are disabled. When you enable them (with the `-a` command line option; see “Enable listings: -a[cdhlns]” on page 15), the initial value of the listing counter is one.

.ln *line-number*

`.ln` is a synonym for `.line`.

.long *expressions*

`.long` is the same as `.int`, see “.int expressions” on page 74.

.macro

The commands `.macro` and `.endm` allow you to define macros that generate assembly output. For example, the following definition specifies a macro `sum` that puts a sequence of numbers into memory.

```
.macro      sum
.long       \from
.if         \to-\from
sum         "(\from+1)",\to
.endif       from=0,to=5
endm
```

With that definition, `SUM 0,5` is equivalent to the following assembly input.

```
.long       0
.long       1
.long       2
.long       3
.long       4
.long       5
```

`.macro` *`macname`*
`.macro` *`macname macargs`*`...`

Begin the definition of a macro called *`macname`*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with *`=deflt`*. For example, the following are all valid `.macro` statements:

`.macro comm`

Begin the definition of a macro called `comm`, which takes no arguments.

`.macro plus1 p, p1`
`.macro plus1 p p1`

Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write `\p` or `\p1` to evaluate the arguments.

`.macro reserve_str p1=0 p2`

Begin the definition of a macro called `reserve_ str`, with two arguments.

The first argument has a default value, but not the second.

After the definition is complete, you can call the macro either as `reserve_str` *`a`*`,` *`b`* (with `\p1` evaluating to *`a`* and `\p2` evaluating to *`b`*), or as `reserve_str,` *`b`* (with `\p1` evaluating as the default, in this case `0`, and `\p2` evaluating to *`b`*).

When you call a macro, you can specify the argument values either by position, or by keyword.
For example, `sum 9,17` is equivalent to `sum to=17, from=9`.

`.endm`

Mark the end of a macro definition.

`.exitm`

Exit early from the current macro definition.

`\@`

`as` maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with `\@`, but *only within a macro definition*.

`.mri` *`val`*

If *`val`* is non-zero, this tells as to enter MRI mode. If *`val`* is zero, this tells `as` to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See “Assemble in MRI compatibility mode: -M” on page 21.

.nolist

Control (in conjunction with the `.list` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

.octa *bignums*

`.octa` This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer.

The term *octa* comes from contexts in which a *word* is two bytes; hence *octa*-word for 16 bytes.

.org *new-lc, fill*

Advance the location counter of the current section to *`new-lc`*. *`new-lc`* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use `.org` to cross sections: if *`new-lc`* has the wrong section, the `.org` directive is ignored. To be compatible with former assemblers, if the section of *`new-lc`* is absolute, as issues a warning, then pretends the section of *`new-lc`* is the same as the current subsection.

`.org` may only increase the location counter, or leave it unchanged; you cannot use `.org` to move the location counter backwards.

Because `as` tries to assemble programs in one pass, *`new-lc`* may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *`fill`* which should be an absolute expression. If the comma and *`fill`* are omitted, *`fill`* defaults to zero.

.p2align[wl] *abs-expr, abs-expr*

`.p2align[wl]` Pad the location counter (in the current subsection) to a particular storage boundary.

The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example `.p2align 3` advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

.psize *lines, columns*

Use this directive to declare the number of lines—and, optionally, the number of columns—to use for each page, when generating listings.

If you do not use `.psize`, listings use a default line-count of 60. You may omit the comma and columns specification; the default width is 200 columns.

`as` generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify *`lines`* as `0`, no formfeeds are generated save those explicitly specified with `.eject`.

.quad *bignums*

`.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum. The term *quad* comes from contexts in which a *word* is two bytes; hence *quad*-word for 8 bytes.

.rept *count*

Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive *`count`* times (where count stands for the appropriate sequence).

```
.rept            3
.long           0
.endr
```

For example, assembling the previous statement is equivalent to assembling the following directive.

```
.long           0
.long           0
.long           0
```

.sbttl "*subheading*"

Use *subheading* as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.scl *class*

Set the storage-class value for a symbol. This directive may only be used inside a `.def`/`.endef` pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

The `.scl` directive is primarily associated with COFF output; when configured to generate `b.out` output format, as accepts this directive but ignores it.

.section *name*

Use the `.section` directive to assemble the following code into a section called *name*.

This directive is only supported for targets that actually support arbitrarily named sections; on `a.out` targets, for example, it is not accepted, even with a standard `a.out` section name.

For COFF targets, the `.section` directive is used in one of the following ways.

```
.section name[,"flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized.

`b`
: bss section (uninitialized data)

`n`
: section is not loaded

`w`
: writable section

`d`
: data section

`r`
: read-only section

`x`
: executable section

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsequent number (see “Sub-sections” on page 46).

For ELF targets, the .section directive is used in the following way.

`.section` *`name`*`[, “`*`flags`*`”[, @`*`type`*`]]`

The optional *`flags`* argument is a quoted string which may contain any combination of the following characters.

`a`
: section is allocatable

`w`
: section is writable

`x`
: section is executable

The optional *`type`* argument may contain one of the following constants.

`@progbits`
section contains data

`@nobits`
section does not contain data (that is, section only occupies space)

If no flags are specified, the default flags depend on the section name. If the section name is not recognized, the default will be for the section to have none of the previously described flags; it will not be allocated in memory, nor will it be writable or executable. The section will contain data.

For ELF targets, the assembler supports another type of `.section` directive for compatibility with the Solaris assembler, as with the following declaration.

`.section “`*`name`*`”[, `*`flags...`*`]]`

Notice that the section name is quoted. There may be a sequence of the following comma-separated flags.

`#alloc`
: section is allocatable

`#write`
: section is writable

`#excinstr`
: section is executable

`.set` *`symbol, expression`*

Set the value of *`symbol`* to *`expression`*. This changes *`symbol`*'s value and type to conform to *`expression`*. If *`symbol`* was flagged as external, it remains flagged. (See "Symbol attributes" on page 57.) You may `.set` a symbol many times in the same assembly.

If you `.set` a global symbol, the value stored in the object file is the last value stored into it.

The syntax for `set` on the HPPA is *`symbol`*`.set` *`expression`*.

`.short` *`expressions`*

`.short` is normally the same as `.word`. See ".word expressions" on page 86.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; for a specific architecture's documentation, see "Machine dependent features" on page 87.

`.single` *`flonums`*

`.single` assembles zero or more flonums, separated by commas. It has the same effect as .float. The exact kind of floating point num-bers emitted depends on how `as` is configured; for a specific architecture's documentation, see "Machine dependent features" on page 87.

`.size`

This directive is generated by compilers to include auxiliary de-bugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. `.size` is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

`.sleb128` *`expressions`*

`.sleb128` stands for "signed little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See also ".uleb128 expressions" on page 86.

.skip *size, fill*

This directive emits *size* bytes, each of value, *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. This is the same as `.space`.

.space *size, fill*

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

WARNING: `.space` has a completely different meaning for HPPA targets; use `.block` as a substitute. See ***HP9000 Series 800 Assembly Language Reference Manual*** (HP 92432-90001) for the meaning of the `.space` directive. See “HPPA dependent features” on page 137.

On the AMD 29K, this directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

WARNING: In most versions of the GNU assembler, the `.space` directive has the effect of `.block`. For each architecture’s documentation, see “Machine dependent features” on page 87.

.stabd, .stabn, .stabs

There are three directives that begin `.stabs`. All emit symbols for use by symbolic debuggers (see “Symbols” on page 34). The symbols are not entered in the `as` hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required, as the following descriptions clarify.

string
: This is the symbol’s name. It may contain any character except `\000`, so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.

type
: An absolute expression. The symbol’s type is set to the low 8 bits of this expression. Any bit pattern is permitted, but `ld` and debuggers choke on silly bit patterns.

other
: An absolute expression. The symbol’s “other” attribute is set to the low 8 bits of this expression.

desc
An absolute expression. The symbol’s descriptor is set to the low 16 bits of this expression.

value
An absolute expression which becomes the symbol’s value.

If a warning is detected while reading a `.stabd`, `.stabn`,or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

`.stabd` *type*, *other* , *desc*
The “name” of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn’t waste space in object files with empty strings.

The symbol’s value is set to the location counter, relocatability. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

`.stabn` *type*, *other* , *desc*, *value*
The name of the symbol is set to the empty string `""`.

`.stabs` *string* , *type*, *other* , *desc*, *value*
All five fields are specified.

.string “*str*”

Copy the characters in *str* to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in “Strings” on page 36

.symver

Use the `.symver` directive to bind symbols to specific version nodes within a source file. This is only supported on ELF platforms, and is typically used when assembling files to be linked into a shared library. There are cases where it may make sense to use this in objects to be bound into an application itself so as to override a versioned symbol from a shared library.

For ELF targets, the `.symver` directive is used like the following declaration shows

`.symver` *name*, *name2@nodename*

In this case, the symbol, *name*, must exist and be defined within the file being assembled.

The `.versym` directive effectively creates a symbol alias with the name *`name2@ nodename`*, and in fact the main reason that we just don't try and create a regular alias is that the `@` character isn't permitted in symbol names. The *`name2`* part of the name is the actual name of the symbol by which it will be externally referenced. The name, *`name`*, itself is merely a name of convenience that is used so that it is possible to have definitions for multiple versions of a function within a single source file, and so that the compiler can unambiguously know which version of a function is being mentioned. The *`nodename`* portion of the alias should be the name of a node specified in the version script supplied to the linker when building a shared library. If you are attempting to override a versioned symbol from a shared library, then *`nodename`* should correspond to the nodename of the symbol you are trying to override.it.

.tag *structname*

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. Tags are used to link structure definitions in the symbol table with instances of those structures. `.tag` is only used when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

.text *subsection*

`.text` tells `as` to assemble the following statements onto the end of the text subsection numbered *`subsection`*, which is an absolute expression. If *`subsection`* is omitted, subsection number zero is used.

.title "*heading*"

Use *`heading`* as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings. `.title` affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.type *int*

This directive, permitted only within `.def`/`.endef` pairs, records the integer *`int`* as the type attribute of a symbol table entry. `.type` is associated only with COFF format output; when `as` is configured for `b.out` output, it accepts this directive but ignores it.

.val *addr*

This directive, permitted only within `.def`/`.endef` pairs, records the address *`addr`* as the value attribute of a symbol table entry.

`.val` is used only for COFF output; when **as** is configured for `b.out`, it accepts this directive but ignores it.

`.uleb128` *expressions*

`uleb128` stands for "unsigned little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See ".sleb128 expressions" on page 82.

`.word` *expressions*

This directive expects zero or more *expressions*, of any section, separated by commas. The size of the number emitted, and its byte order, depend on what target computer for which the assembly builds.

WARNING: To support compilers on machines with a 32-bit address space, having less than 32-bit addressing, this requires special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it), ignore this issue. For documentation for specific processors, see "Machine dependent features" on page 87. In order to assemble compiler output into something that works, `as` occasionally does strange things to `.word` directives.

Directives of the form, `.word sym1-sym2`, are often emitted by compilers as part of jump tables. Therefore, when `as` assembles a directive of the form, `.word sym1-sym2`, and the difference between `sym1` and `sym2` does not fit in 16 bits, `as` creates a *secondary jump table*, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to `sym2`. The original `.word` contains `sym1` minus the address of the long-jump to `sym2`. If there were several occurrences of `.word sym1-sym2` before the secondary jump table, all of them are adjusted.

If there was a `.word sym3-sym4` that also did not fit in sixteen bits, a long-jump to `sym4` is included in the secondary jump table, and the `.word` directives are adjusted to contain `sym3` minus the address of the long-jump to `sym4`; and so on, for as many entries in the original jump table as necessary.

9

Machine dependent features

The machine instruction sets are (almost by definition) different on each machine where the GNU assembler, `as`, runs. Floating point representations vary as well, and as often supports a few additional directives or command-line options for compatibility with other assemblers on a particular platform. Finally, some versions of `as` support special pseudo-instructions for branch optimization.

There is discussion on most of these differences, though there aren't details on any machine's instruction set. For details on that subject, see the specific hardware manufacturer's documentation.

The following architectures are discussed.

- "AMD 29K dependent features" on page 89
- "ARC dependent features" on page 95
- "ARM dependent features" on page 99
- "AT&T/Intel dependent features" on page 103
- "D10V dependent features" on page 113
- "H8/300 dependent features" on page 123
- "H8/500 dependent features" on page 131
- "HPPA dependent features" on page 137
- "SH dependent features" on page 143

10

AMD 29K dependent features

The following documentation discusses the features pertinent to the AMD 29K regarding the GNU assembler.

AMD 29K options

`as` has no additional command-line options for the AMD 29K family.

AMD 29K syntax

The following documentation discusses the features pertinent to the AMD 29K regarding the syntax for the GNU assembler.

AMD 29K macros

The macro syntax used on the AMD 29K is like syntax in the AMD 29K Family Macro Assembler Specification. Normal `as` macros should still work.

AMD 29K special characters

`;` is the line comment character.

The question mark character, `?`, is permitted in identifiers (but *may not begin* an identifier).

AMD 29K register names

General-purpose registers are represented by predefined symbols of the form `GR`*nnn* (for global registers) or `LR`*nnn* (for local registers), where *nnn* represents a number between `0` and `127`, written with no leading zeros. The leading letters may be in either upper or lower case; for example, `gr13` and `LR7` are both valid register names.

You may also refer to general-purpose registers by specifying the register number as the result of an expression (prefixed with `%%` to flag the expression as a register number), as in the following example.

```
%% expression
```

expression in the previous example's case must be an absolute expression evaluating to a number between `0` and `255`. The range [0, 127] refers to global registers, and the range [128, 255] to local registers.

In addition, `as` understands the protected special-purpose register names for the AMD 29K family, as in the following.

```
vab      chd       pc0
ops      chc       pc1
cps      rbp       pc2
cfg      tmc       mmu
cha      tmr       lru
```

These unprotected special-purpose register names are also recognized:

```
ipc      alu     fpe
ipa      bp      inte
ipb      fc      fps
q        cr      exop
```

AMD 29K floating point

The AMD 29K family uses IEEE floating-point numbers.

AMD 29K machine directives

The following documentation discusses the features pertinent to the AMD 29K regarding the machine directives for the GNU assembler.

`.block` *`size`*, *`fill`*
: This directive emits *`size`* bytes, each of value *`fill`*. Both *`size`* and *`fill`* are absolute expressions. If the comma and *`fill`* are omitted, *`fill`* is assumed to be zero. In other versions of the GNU assembler, this directive is called `.space`.

`.cputype`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.file`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

WARNING: In other versions of the GNU assembler, `.file` is used for the directive called `.app-file` in the AMD 29K support.

`.line`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.sect`
: This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.use section` *`name`*
: Establishes the section and subsection for the following code; section name may be one of `.text`, `.data`, `.data1`, or `.lit`. With one of the first three *`section name`* options, `.use` is equivalent to the machine directive, *`section name`*; the remaining case, `.use .lit`, is the same as `.data 200`.

AMD 29K opcodes

`as` implements all the standard AMD 29K opcodes. No additional pseudo-instructions are needed on this family.

For information on the 29K machine instruction set, see ***AMD 29000 User's Manual***, Advanced Micro Devices, Inc.

11

ARC dependent features

The following documentation discusses the features pertinent to the ARC processor regarding the GNU assembler.

ARC command-line options

The ARC chip family includes several successive levels (or other variants) of chip, using the same core instruction set, but including a few additional instructions at each level.

By default, `as` assumes the core instruction set (ARC base). The `.cpu` pseudo-op is intended to be used to select the variant.

`-mbig-endian`
`-mlittle-endian`
: Any ARC configuration of `as` can select big-endian or little-endian output at run time (unlike most other GNU development tools, which must be configured for one or the other). Use '`-mbig-endian`' to select big-endian output, and '`-mlittle-endian`' for little-endian.

ARC floating point

The ARC processor family currently does not have hardware floating point support. Software floating point support is provided by `gcc` and uses IEEE floating-point numbers.

ARC machine directives

The ARC version of `as` supports the following additional machine directive.

`.cpu`
: This must be followed by the desired CPU. The ARC is intended to be customizable, `.cpu` is used to select the desired variant (although, currently, there are none).

12

ARM dependent features

The following documentation discusses the features pertinent to the ARM processor regarding the GNU assembler.

ARM options

`as` has no additional command-line options for the ARM processor family.

ARM syntax

`as` has no additional command-line options for the ARM processor family.

ARM special characters

'`;`' is the line comment character.

ARM floating point

The ARM family uses IEEE floating-point numbers.

ARM machine directives

The following assembler directives are for the ARM processor.

`.code [16| 32]`

This directive selects the instruction set being generated.

The value 16 selects Thumb, with the value 32 selecting ARM.

`.thumb`

This performs the same action as `.code 16`.

`.arm`

This performs the same action as `.code 32`.

ARM opcodes

as implements all the standard ARM opcodes.

For information on the ARM or Thumb instruction sets, see ***ARM Software Development Toolkit Reference Manual***, Advanced RISC Machines, Ltd.

13

AT&T/Intel dependent features

The following documentation discusses the features pertinent to the AT&T/Intel processors regarding the GNU assembler.

80386 options

The 80386 has no machine dependent options.

AT&T syntax versus Intel syntax

In order to maintain compatibility with the output of GCC, as supports AT&T System V/386 assembler syntax. This is quite different from Intel syntax. We mention these differences because almost all 80386 documents used only Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by `$`; Intel immediate operands are undelimited (Intel `push 4` is AT&T `pushl $4`). AT&T register operands are preceded by `%`; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) `jump/call` operands are prefixed by `*`; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel `add eax, 4` is `addl $4, %eax`. The `source, dest` convention is maintained for compatibility with previous Unix assemblers.
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of `b`, `w`, and `l` specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixes memory operands (*not* the opcodes themselves) with `byte ptr`, `word ptr`, and `dword ptr`. Thus, Intel `mov al, byte ptr foo` is `movb foo, %al` in AT&T syntax.
- Immediate form long jumps and calls are `lcall/ljmp $ section, $offset` in AT&T syntax; the Intel syntax is `call/jmp far section: offset`. Also, the `far return` instruction is `lret $stack-adjust` in AT&T syntax; Intel syntax is `ret far stack-adjust`.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

AT&T opcode naming

Opcode names are suffixed with one character modifiers that specify the size of operands.

The letters `b`, `w`, and `l` specify byte, word, and long operands. If no suffix is specified by an instruction and it contains no memory operands then `as` tries to fill in the missing suffix based on the destination register operand (the last one by convention).

Thus, `mov %ax, %bx` is equivalent to `movw %ax, %bx`; also, `mov $1, %bx` is equivalent to `movw $1, %bx`.

NOTE: This is incompatible with the AT&T Unix assembler which assumes that a missing opcode suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the opcode suffix.)

Almost all opcodes have the same names in AT&T and Intel format. There are a few exceptions.

The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend from and a size to zero extend *to*. This is accomplished by using two opcode suffixes in AT&T syntax.

Base names for `sign extend` and `zero extend` are `movs...`and `movz...` in AT&T syntax (`movsx` and `movzx` in Intel syntax).

The opcode suffixes are tacked on to this base name, the *from* suffix before the *to* suffix. Thus, `movsbl %al, %edx` is AT&T syntax for “move sign extend *from* `%al` *to* `%edx`.” Possible suffixes, thus, are `bl` (from byte to long), `bw` (from byte to word), and `wl` (from word to long).

The following Intel-syntax conversion instructions are called `cbtw`, `cwtl`, `cwtd`, and `cltd` in AT&T naming. **`as`** accepts either naming for the following instructions.

- `cbw`
  sign-extend byte in `%al` to word in `%ax`
- `cwde`
  sign-extend word in `%ax` to long in `%eax`
- `cwd`
  sign-extend word in `%ax` to long in `%dx:%ax`
- `cdq`
  sign-extend dword in `%eax` to quad in `%edx:%eax`

Far call/jump instructions are `lcall` and `ljmp` in AT&T syntax, but `call far` and `jump far` in Intel convention.

80386 register naming

Register operands are always prefixes with **`%`**.

The 80386 registers consist of the following register operands.

- 8 32-*bit registers*:

  `%eax` (the accumulator), `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp` (the frame pointer), and `%esp` (the stack pointer).
- 8 16-*bit low-ends*:

  `%ax`, `%bx`, `%cx`, `%dx`, `%di`, `%si`, `%bp`, and `%sp`.
- 8 8-*bit registers*

`%ah`, `%al`, `%bh`, `%bl`, `%ch`, `%cl`, `%dh`, and `%dl`. (These are the high-bytes and low-bytes of `%ax`, `%bx`, `%cx`, and `%dx`.)

- 6 *section registers*:

  `%cs` (*code section*), `%ds` (*data section*), `%ss` (*stack section*), `%es`, `%fs`, and `%gs`.
- 3 *processor control registers*:

  `%cr0`, `%cr2`, and `%cr3`.
- 6 *debug registers*:

  `%db0`, `%db1`, `%db2`, `%db3`, `%db6`, and `%db7`.
- 2 *test registers*:

  `%tr6` and `%tr7`.
- 8 *floating point register stack*:

  `%st`, or, equivalently, `%st(0)`, `%st(1)`, `%st(2)`, `%st(3)`, `%st(4)`, `%st(5)`, `%st(6)`, and `%st(7)`.

80386 opcode prefixes

Opcode prefixes are used to modify the following opcode. They are used to repeat string instructions, to provide section overrides, to per-form bus lock operations, and to give operand and address size (16-bit operands are specified in an instruction by prefixing what would nor-mally be 32-bit operands with a *operand size* opcode prefix). Opcode prefixes are usually given as single-line instructions with no operands, and must directly precede the instruction they act upon. For example, the '`scas`' (scan string) instruction is repeated with the following.

```
repne
scas
```

The following is a list of opcode prefixes:

- *Section override prefixes*:

  `cs`, `ds`, `ss`, `es`, `fs` and `gs`. These are automatically added by using the *`section:memory-operand`* form for memory references.
- *Operand/Address size prefixes*:

  `data16` and `addr16` change 32-*bit operands/addresses* into 16-*bit operands/addresses*.

  Note:16-*bit addressing* modes are not yet supported (i.e., 8086 and 80286 addressing modes).
- The *bus lock prefix*, `lock`, inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).

- The *wait for coprocessor* prefix, `wait`, waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.
- The `rep`, `repe`, and `repne` prefixes are added to string instructions to make them repeat `%ecx` times.

AT&T memory references

An Intel syntax indirect memory reference of the following form translates into the AT&T syntax like the following declaration.

```
section:[base + index*scale + disp]
```

It then outputs like the following example.

```
section:disp(base, index, scale)
```

base and *index* are the optional 32-bit base and index registers, *disp* is the optional displacement, and *scale*, taking the values 1, 2, 4, and 8, multiplies *index* to calculate the address of the operand. If no *scale* is specified, *scale* is taken to be 1. *section* specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Section overrides in AT&T syntax *must* have be preceded by a `%`. If you specify a section override which coincides with the default section register, `as` does *not* output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

AT&T: '`-4(%ebp)`', Intel: '`[ebp - 4]`'
: *base* is '`%ebp`'; *disp* is '-4'. *section* is missing, and the default section is used ('`%ss`' for addressing with '`%ebp`' as the base register). *index*, *scale* are both missing.

AT&T: '`foo(,%eax,4)`', Intel: '`[foo + eax*4]`'
: *index* is '`%eax`' (scaled by a scale4); *disp* is '`foo`'. All other fields are missing. The section register here defaults to '`%ds`'.

AT&T: '`foo(,1)`'; Intel '`[foo]`'
: This uses the value pointed to by '`foo`' as a memory operand. Note that *base* and *index* are both missing, but there is only *one* '`,`'. This is a syntactic exception.

AT&T: '`%gs:foo`'; Intel '`gs:foo`'
: This selects the contents of the variable '`foo`' with section register *section* being '`%gs`'.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with `*`. If no `*` is specified, `as` always chooses PC relative addressing for jump/call labels.

Any instruction that has a memory operand *must* specify its size (byte, word, or long) with an opcode suffix (`b`, `w`, or `l`, respectively).

Handling of Jump instructions for 80386 machines

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e., prefixing the jump instruction with the `addr16` opcode prefix), since the 80386 insists upon masking `%eip` to 16 bits after the word displacement is added.

NOTE: ‘`jcxz`’, ‘`jecxz`’, ‘`loop`’, ‘`loopz`’, ‘`loope`’, ‘`loopnz`’ and ‘`loopne`’ instructions only come in byte displacements, so that if you use these instructions (GCC does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding `jcxz foo` to the following.

```
             jcxz cx_zero
             jmp cx_nonzero
cx_zero:     jmp foo
cx_nonzero:
```

AT&T floating point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty).

These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an opcode suffix and a constructor associated with it. Opcode suffixes specify operand’s data types. Constructors build these data types into memory.

- ***Floating point constructors*** are:

  `.float` or `.single`, `.double`, and `.tfloat` for 32-, 64-, and 80-bit formats. These correspond to opcode suffixes, `s`, `l`, and `t`. `t` stands for *temporary real*, and that the 80387 only supports this format via the `fldt` (*load temporary real* to *stack top*) and `fstpt` (*store temporary real* and *pop stack*) instructions.

- ***Integer constructors*** are:

  `.word`, `.long` or `.int`, and `.quad` for the 16-, 32-, and 64-bit integer formats.

The corresponding *opcode suffixes* are `s` (single), `l` (long), and `q` (quad). As with the *temporary real* format, the 64-bit `q` format is only present in the `fildq` (*load quad integer* to *stack top*) and `fistpq` (*store quad integer* and *pop stack*) instructions.

Register to register operations do not require opcode suffixes, so that the statement, `fst %st, %st(1)`, is equivalent to `fstl %st, %st(1)`.

Since the 80387 automatically synchronizes with the 80386, `fwait` instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, `as` suppresses the `fwait` instruction whenever it is implicitly selected by one of the `fn` the instructions. For example, `fsave` and `fnsave` are treated identically. In general, all the `fn` the instructions are made equivalent to `f` the instructions. If `fwait` is desired, it must be explicitly coded.

Writing 16-bit Code for 80386 machines

While GAS normally writes only *pure* 32-bit i386 code, it has limited support for writing code to run in real mode or in 16-bit protected mode code segments. To do this, insert a `.code16` directive before the assembly language instructions to be run in 16-bit mode. You can switch GAS back to writing normal 32-bit code with the `.code32` directive.

GAS understands exactly the same assembly language syntax in 16- bit mode as in 32-bit mode. The function of any given instruction is exactly the same regardless of mode, as long as the resulting object code is executed in the mode for which GAS wrote it. So, for example, the `ret` mnemonic produces a 32-bit return instruction regardless of whether it is to be run in 16-bit or 32-bit mode. (If GAS is in 16-bit mode, it will add an operand size prefix to the instruction to force it to be a 32-bit return.)

This means, for one thing, that you can use GNU CC to write code to be run in real mode or 16-bit protected mode. Just insert the statement `asm(".code16");` at the beginning of your C source file, and while GNU CC will still be generating 32-bit code, GAS will automatically add all the necessary size prefixes to make that code run in 16-bit mode. Of course, since GNU CC only writes small-model code (it doesn't know how to attach segment selectors to pointers like native x86 compilers do), any 16-bit code you write with GNU CC will essentially be limited to a 64K address space. Also, there will be a code size and performance penalty due to all the extra address and operand size prefixes GAS has to add to the instructions.

NOTE: Placing GAS in 16-bit mode does not mean that the resulting code will necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you would have to refrain from using *any* 32-bit constructs which require GAS to output address or operand size prefixes. At the moment this would be rather difficult, because GAS currently supports *only* 32-bit addressing modes: when writing 16-bit code, it *always* outputs address size prefixes for any instruction that uses a non-register addressing mode. So you can write code that runs on 16-bit processors, but only if that code never references memory.

Notes on AT& T and the 80386 instructions

There is some trickery concerning the `mul` and `imul` instructions that deserves mention. The 16-, 32-, and 64-bit expanding multiplies (base opcode `0xf6`; extension 4 for `mul` and 5 for `imul`) can be output only in the one operand form. Thus, `imul %ebx, %eax` does *not* select the expanding multiply; the expanding multiply would clobber the `%edx` register, and this would confuse gcc output. Use `imul %ebx` to get the 64-bit product in `%edx:%eax`.

We have added a two operand form of `imul` when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying `%eax` by 69, for example, can be done with `imul $69, %eax` rather than `imul $69, %eax, %eax`.

14

D10V dependent features

The following documentation discusses the features pertinent to the D10V processor regarding the GNU assembler.

D10V options

The Mitsubishi D10V version of `as` uses the following machine dependent option.

`-O`
: The D10V can often execute two sub-instructions in parallel.

When this option is used, `as` will attempt to optimize its output by detecting when instructions can be executed in parallel.

D10V syntax

The D10V syntax is based on the syntax in the Mitsubishi document, ***D10V Architecture: A VLIW Mediaprocessor for Multimedia Applications*** (Mitsubishi Electric Corporation, version 1.00; October 24, 1996). The differences are detailed in the following discussions.

D10V size modifiers

The D10V version of **`as`** uses the instruction names in the Mitsubishi document, ***D10V Architecture: A VLIW Mediaprocessor for Multimedia Applications*** (Mitsubishi Electric Corporation, version 1.00; October 24, 1996).. However, the names in the documentation are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. How does the assembler pick the correct form? **`as`** will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either '`.s`' (short) or '`.l`' (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write '`bra.s foo`'. **`objdump`** and **`gdb`** will always append '`.s`' or '`.l`' to instructions which have both short and long forms.

D10V sub-instructions

The D10V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described in the next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols described in the following documentation; see "D10V special characters" on page 116.

D10V special characters

'`;`' and '`#`' are the line comment characters. Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially. To specify the executing order, use the following symbols.

`->`
: Sequential with instruction on the left first.

`<-`
: Sequential with instruction on the right first.

`||`
: Parallel.

The D10V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line, as the following documentation discusses.

`abs a1 -> abs r0`
: Execute these sequentially. The instruction on the right is in the right container and is executed second.

`abs r0 <- abs a1`
: Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

`ld2w r2,@r8+ || mac a0,r0,r7`
: Execute these in parallel.

```
ld2w r2,@r8+ ||
mac a0,r0,r7
```
: Two-line format. Execute these in parallel.

```
ld2w r2,@r8+
mac a0,r0,r7
```
: Two-line format. Execute these sequentially. Assembler will put them in the proper containers.

```
ld2w r2,@r8+ ->
mac a0,r0,r7
```
: Two-line format. Execute these sequentially. Same as previously described instructions except second instruction will always go into right container.

Since '`$`' has no special meaning, you can use it in symbol names.

D10V register names

You can use the predefined symbols 'r0' through 'r15' to refer to the D10V registers. You can also use 'sp' as an alias for 'r15'. The accumulators are 'a0' and 'a1'. There are special register-pair names that may optionally be used in opcodes that require even-numbered registers. Register names are not case sensitive.

```
r0-r1
r2-r3
r4-r5
r6-r7
r8-r9
r10-r11
r12-r13
r14-r15
```

The D10V also has predefined symbols for the following control registers and status bits.

`psw`	Processor Status Word
`bpsw`	Backup Processor Status Word
`pc`	Program Counter
`bpc`	Backup Program Counter
`rpt_c`	Repeat Count
`rpt_s`	Repeat Start address
`rpt_e`	Repeat End address
`mod_s`	Modulo Start address
`od_e`	Modulo End address
`iba`	Instruction Break Address
`f0`	Flag 0
`f1`	Flag 1
`c`	Carry flag

D10V addressing modes

`as` understands the following addressing modes for the D10V. `R`*`n`* in the following documentation refers to any of the numbered registers, but not the control registers.

`R`*`n`*
: Register direct

`@R`*`n`*
: Register indirect

`@R`*`n`*`+`
: Register indirect with post-increment

`@R`*`n`*`-`
: Register indirect with post-decrement

`@-SP`
: Register indirect with pre-decrement

`@(`*`disp`*`,R`*`n`*`)`
: Register indirect with displacement

`addr`
: PC relative address (for branch or rep).

`#imm`
: Immediate data (the ‘`#`’ is optional and ignored)

`@WORD` modifier for D10V

Any symbol followed by `@word` will be replaced by the symbol’s value shifted right by 2. This is used in situations such as loading a register with the address of a function (or any other code fragment). For example, if you want to load a register with the location of the function `main` then jump to that function, you could do it as follows.

```
ldi r2, main @WORD
jmp r2
```

D10V floating point

The D10V has no hardware floating point, but the `.float` and `.double` directives generates IEEE floating-point numbers for compatibility with other development tools.

D10V opcodes

For detailed information on the D10V machine instruction set, see the Mitsubishi document, ***D10V Architecture: A VLIW Mediaprocessor for Multimedia Applications*** (Mitsubishi Electric Corporation, version 1.00; October 24, 1996); **as** implements all the standard D10V opcodes found there. The only changes are those described in the section on size modifiers.

15

H8/300 dependent features

The following documentation discusses the features pertinent to the H8/300 processor regarding the GNU assembler.

H8/300 options

`as` has no additional command-line options for the Hitachi H8/300 family.

H8/300 syntax

The following documentation discusses the features pertinent to the H8/300 regarding the syntax for the GNU assembler.

Special characters

`;` is the line comment character.

`$` can be used instead of a newline to separate statements. Therefore, *you may not use* `$` *in symbol names* on the H8/300.

H8/300 register nnames

You can use predefined symbols of the form `r`*n*`h` and `r`*n*`l` to refer to the H8/300 registers as sixteen 8-bit general-purpose registers. *n* is a digit from `0` to `7`); for instance, both `r0h` and `r7l` are valid register names.

You can also use the eight predefined symbols `r`*n* to refer to the H8/300 registers as 16-bit registers (you must use this form for addressing).

On the H8/300H, you can also use the eight predefined symbols `er`*n* (`er0...er7`) to refer to the 32-bit general purpose registers.

The two control registers are called `pc` (program counter; a 16-bit register, except on the H8/300H where it is 24 bits) and `ccr` (condition code register; an 8-bit register). `r7` is used as the stack pointer, and can also be called `sp`.

H8/300 addressing modes

`as` understands the following addressing modes for the H8/300.

`r`*n*
: Register direct

`@r`*n*
: Register indirect

`@(`*d*`, r`*n*`)`
`@(`*d*`:16, r`*n*`)`
`@(`*d*`:24, r`*n*`)`
: Register indirect: 16-bit or 24-bit displacement, *d*, from register, *n*. (24-bit displacements are only meaningful on the H8/300H.)

`@r`*n*`+`
: Register indirect with post-increment.

`@-r`*n*
: Register indirect with pre-decrement.

`@`*aa*
`@`*aa*`:8`
`@`*aa*`:16`
`@`*aa*`:24`
: Absolute address `aa`. (The address size `:24` only makes sense on the H8/300H.)

`#`*xx*
`#`*xx*`:8`
`#`*xx*`:16`
`#`*xx*`:32`
: Immediate data *xx*. You may specify the '`:8`', '`:16`', or '`:32`' for clarity, if you wish; but `as` neither requires this nor uses it—the data size required is taken from context.

`@@`*aa*
`@@`*aa*`:8`
: Memory indirect. You may specify the '`:8`' for clarity, if you wish; but `as` neither requires this nor uses it.

H8/300 floating point

The H8/300 family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

H8/300 machine directives

`as` has only the following machine-dependent directive for the H8/300.

`.h8300h`

Recognize and emit additional instructions for the H8/300H variant, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

On the H8/300 family (including the H8/300H) `.word` directives generate 16-bit numbers.

H8/300 opcodes

For detailed information on the H8/300 machine instruction set, see ***H8/300 Series Programming Manual*** (Hitachi ADE–602–025).

For information specific to the H8/300H, see ***H8/300H Series Programming Manual*** (Hitachi).

`as` implements all the standard H8/300 opcodes. No additional pseudo-instructions are needed on this family.

Four H8/300 instructions (`add`, `cmp`, `mov`, `sub`) are defined with variants using the suffixes `.b`, `.w`, and `.l` to specify the size of a memory operand.

`as` supports these suffixes, but does not require them; since one of the operands is always a register, `as` can deduce the correct size.

For instance, since `r0` refers to a 16-bit register, note the distinctions in the following examples.

```
mov    r0,@foo

mov    w r0,@foo
```

If you use the size suffixes, `as` issues a warning when the suffix and the register size do not match.

16

H8/500 dependent features

The following documentation discusses the features pertinent to the H8/500 processor regarding the GNU assembler.

H8/500 options

`as` has no additional command-line options for the Hitachi H8/500 family.

H8/500 syntax

The following documentation discusses the features pertinent to the H8/500 regarding the syntaxx for the GNU assembler.

H8/500 special characters

`!` is the line comment character.

`;` can be used instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

H8/500 register names

You can use the predefined symbols `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, and `r7` to refer to the H8/500 registers.

The H8/500 also has the following control registers.

`cp`	code pointer
`dp`	data pointer
`bp`	base pointer
`tp`	stack top pointer
`ep`	extra pointer
`sr`	status register
`ccr`	condition code register

All registers are 16 bits long. To represent 32 bit numbers, use two adjacent registers; for distant memory addresses, use one of the segment pointers (`cp` for the program counter; `dp` for `r0–r3`; `ep` for `r4` and `r5`; and `tp` for `r6` and `r7`.

H8/500 addressing modes

`as` understands the following addressing modes for the H8/500:

`R`*`n`*

Register direct.

`@R`*`n`*

Register indirect.

`@(d:8, Rn)`
: Register indirect with 8 bit signed displacement.

`@(d:16, Rn)`
: Register indirect with 16 bit signed displacement.

`@-Rn`
: Register indirect with pre-decrement.

`@Rn+`
: Register indirect with post-increment.

`@aa:8`
: 8 bit absolute address.

`@aa:16`
: 16 bit absolute address.

`#xx:8`
: 8 bit immediate.

`#xx:16`
: 16 bit immediate.

H8/500 floating point

The H8/500 family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

H8/500 machine directives

`as` has no machine-dependent directives for the H8/500. However, on this platform the `.int` and `.word` directives generate 16-bit numbers.

H8/500 opcodes

For detailed information on the H8/500 machine instruction set, see H8***/500 Series Programming Manual*** (Hitachi M21T001).

`as` implements all the standard H8/500 opcodes. No additional pseudo-instructions are needed on this family.

17

HPPA dependent features

The following documentation discusses the features pertinent to the HPPA processor regarding the GNU assembler.

As a back end for GNU CC, `as` has been thoroughly tested and should work extremely well for the HPPA targets. We have tested it only minimally on hand-written assembly code and no one has tested it much on the assembly output from the HP compilers.

The format of the debugging sections has changed since the original `as` port (version 1.3*x*) was released; therefore, you must rebuild all HPPA objects and libraries with the new assembler so that you can debug the final executable.

The HPPA `as` port generates a small subset of the relocations available in the SOM and ELF object file formats. Additional relocation support will be added as it becomes necessary.

HPPA options

`as` has no machine-dependent command-line options for the HPPA.

HPPA syntax

The assembler syntax closely follows the HPPA instruction set reference manual; assembler directives and general syntax closely follow the HPPA assembly language reference manual, with a few noteworthy differences.

First, a colon may immediately follow a label definition. This is simply for compatibility with how most assembly language programmers write code.

Some obscure expression parsing problems may affect hand written code which uses the `spop` instructions, or code which makes significant use of the `!` line separator.

`as` is much less forgiving about missing arguments and other similar oversights than the HP assembler. `as` notifies you of missing arguments as syntax errors; this is regarded as a feature, not a bug.

Finally, `as` allows you to use an external symbol without explicitly importing the symbol.

WARNING: In the future this allowance will be an error for HPPA targets.

Special characters for HPPA targets include the following.

`;` is the line comment character.

`!` can be used instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

HPPA floating point

The HPPA family uses IEEE floating-point numbers.

HPPA assembler directives

`as` for the HPPA supports many additional directives for compatibility with the native assembler. The following documentation only briefly describes them. For detailed information on HPPA-specific assembler directives, see ***HP9000 Series 800 Assembly Language Reference Manual*** (HP 92432-90001).

`as` does *not* support the following assembler directives described in the HP manual:

```
.endm        listoff      macro
.enter       liston
.leave       locct
```

Beyond those implemented for compatibility, `as` supports one additional assembler directive for the HPPA: `.param`. It conveys register argument locations for static functions. Its syntax closely follows the `.export` directive.

The following are the additional directives in `as` for the HPPA:

`.block` *`n`*
`.blockz` *`n`*
: Reserve *`n`* bytes of storage, and initialize them to zero.

`.call`
: Mark the beginning of a procedure call. Only the special case with no arguments is allowed.

`.callinfo [`*`param`*`=`*`value`*`, ...][`*`flag`*`, ...]`
: Specify a number of parameters and flags that define the environment for a procedure. *`param`* may be any of *`frame`* (frame size), `entry_gr` (end of general register range), `entry_fr` (end of float register range), `entry_sr` (end of space register range). The values for *`flag`* are `calls` or `caller` (proc has subroutines), `no_calls` (proc does not call subroutines), `save_rp` (preserve return pointer), `save_sp` (proc preserves stack pointer), `no_unwind` (do not unwind this proc), `hpux_int` (proc is interrupt routine).

`.code`
: Assemble into the standard section called `$TEXT$`, subsection `$CODE$`.

`.copyright “`*`string`*`”`
: In the SOM object format, insert *`string`* into the object code, marked as a copyright string.

`.enter`
: Not yet supported; the assembler rejects programs containing this directive.

`.entry`
: Mark the beginning of a procedure.

`.exit`
: Mark the end of a procedure.

`.export` *`name`*`[,`*`typ`*`][,`*`param`*`=`*`r`*`]`
: Make a procedure *`name`* available to callers. *`typ`*, if present, must be one of `absolute`, `code` (ELF only, not SOM), `data`, `entry`, `data`, `entry`, `millicode`, `plabel`, `pri_prog`, or `sec_prog`.

  `param`, if present, provides either relocation information for the procedure arguments and result, or a privilege level. *`param`* may be `argw` *`n`* (where *`n`* ranges from *`0`* to *`3`*, and indicates one of four one-word arguments); `rtnval` (the proce-dure’s result); or `priv_lev` (privilege level). For arguments or the result, *`r`* specifies how to relocate, and must be one of `no` (not relocatable), `gr` (argument is in general register), `fr` (in floating point register), or ‘`fu`’ (upper half of float register). For `priv_lev`, *`r`* is an integer.

`.half` *n*
: Define a two-byte integer constant *n*; synonym for the portable `as` directive, `.short`.

`.import` *name*`[,`*typ*`]`
: Converse of `.export`; make a procedure available to call. The arguments use the same conventions as the first two arguments for `.export`.

`.label` *name*
: Define *name* as a label for the current assembly location.

`.leave`
: Not yet supported; the assembler rejects programs containing this directive.

`.origin` *lc*
: Advance location counter to *lc*. Synonym for the `{No value for ''as''}` portable directive `.org`.

`.param` *name*`[,`*typ*`][,`*param*`=`*r*`]`
: Similar to `.export`, but used for static procedures.

`.proc`
: Use preceding the first statement of a procedure.

`.procend`
: Use following the last statement of a procedure.

`label.reg expr`
: Synonym for `.equ`; define *label* with the absolute expression *expr* as its value.

`.space secname[,params]`
: Switch to section *secname*, creating a new section by that name if necessary. You may only use *params* when creating a new section, not when switching to an existing one. *secname* may identify a section by number rather than by name. If specified, the list *params* declares attributes of the section, identified by keywords. The keywords recognized are `spnum=`*exp* (identify this section by the number *exp*, an absolute expression), `sort=`*exp* (order sections according to this sort key when linking; *exp* is an absolute ex-pression), `unloadable` (section contains no loadable data), `notdefined` (this section defined elsewhere), and `private` (data in this section not available to other programs).

`.spnum` *secnam*
: Allocate four bytes of storage, and initialize them with the section number of the section named *secnam*. (You can define the section number with the HPPA `.space` directive.)

`.string` *"str"*
: Copy the characters in the string *str* to the object file. See "Strings" on page 36 for information on escape sequences you can use in `as` strings.

WARNING: The HPPA version of `.string` differs from the usual `as` definition: it does *not* write a zero byte after copying `str`.

`.stringz "str"`
: Like `.string`, but appends a zero byte after copying `str` to object file.

`.subspa name[,params]`

`.nsubspa name[,params]`
: Similar to `.space`, but selects a subsection name within the current section. You may only specify `params` when you create a subsection (in the first instance of `.subspa` for this `name`). If specified, the list `params` declares attributes of the subsection, identified by keywords. The keywords recognized are `quad=expr` ("quadrant" for this subsection), `align=expr` (alignment for beginning of this subsection; a power of two), `access=expr` (value for "access rights" field), `sort=expr` (sorting order for this subspace in link), `code_only` (subsection contains only code), `unloadable` (subsection cannot be loaded into memory), `common` (subsection is common block), `dup_comm` (initialized data may have duplicate names), or `zero` (subsection is all zeros, do not write in object file).

  `.nsubspa` always creates a new subspace with the given name, even if one with the same name already exists.

`.version "str"`
: Write `str` as version identifier in object code.

HPPA opcodes

For detailed information on the HPPA machine instruction set, see ***PA-RISC Architecture and Instruction Set Reference Manual*** (HP 09740- 90039).

18

SH dependent features

The following documentation discusses the features pertinent to the Hitachi SH processor regarding the GNU assembler.

Hitachi SH options

`as` has no additional command-line options for the Hitachi SH family.

Hitachi SH syntax

The following documentation discusses the features pertinent to the Hitachi SH regarding the syntax for the GNU assembler.

Hitachi SH special characters

`!` is the line comment character.

You can use `;` instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

Hitachi SH register names

You can use the predefined symbols '`r0`', '`r1`', '`r2`', '`r3`', '`r4`', '`r5`', '`r6`', '`r7`', '`r8`', '`r9`', '`r10`', '`r11`', '`r12`', '`r13`', '`r14`', and '`r15`' to refer to the SH registers.

The SH also has the following control registers.

`pr`
: procedure register (holds return address)

`pc`
: program counter

`mach`
`macl`
: high and low multiply accumulator registers

`sr`
: status register

`gbr`
`global`
: base register

`vbr`
: vector base register (for interrupt vectors)

Hitachi SH addressing modes

`as` understands the following addressing modes for the SH. `R`*n* in the following refers to any of the numbered registers, but not the control registers.

`R`*n*
: Register direct.

`@R`*`n`*
: Register indirect.

`@-R`*`n`*
: Register indirect with pre-decrement.

`@R`*`n`*`+`
: Register indirect with post-increment.

`@(`*`disp`*`, R`*`n`*`)`
: Register indirect with displacement.

`@(R0, R`*`n`*`)`
: Register indexed.

`@(`*`disp`*`, GBR)`
: GBR offset.

`@(R0, GBR)`
: GBR indexed.

`addr`
`@(`*`disp`*`, PC)`
: PC relative address (for branch or for addressing memory). The `as` implementation allows you to use the simpler form *`addr`* anywhere a PC relative address is called for; the alternate form is supported for compatibility with other assemblers.

`#`*`imm`*
: Immediate data.

Hitachi SH floating point

The SH family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

Hitachi SH machine directives

`as` has no machine-dependent directives for the SH.

Hitachi SH opcodes

For detailed information on the SH machine instruction set, see ***SH-Microcomputer User's Manual*** (Hitachi Micro Systems, Inc.).

`as` implements all the standard SH opcodes. No additional pseudo-instructions are needed on this family. Note, however, that because `as` supports a simpler form of PC-relative addressing, you may simply write the following (for example).

```
mov.l bar,r0
```

Other assemblers might require an explicit displacement to `bar` from the program counter:

```
mov.l @(disp, PC)
```

19

Intel 960 dependent features

The following documentation discusses the features pertinent to the Intel 960 processor regarding the GNU assembler.

i960 command-line options

`-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC`

Select the 80960 architecture. Instructions or features not supported by the selected architecture cause fatal errors.

`-ACA` is equivalent to `-ACA_A`; `-AKC` is equivalent to `-AMC`.

Synonyms are provided for compatibility with other tools.

If you do not specify any of these options, `as` generates code for any instruction or feature that is supported by *some* version of the i960 (even if this means mixing architectures!).

In principle, `as` attempts to deduce the minimal sufficient processor type if none is specified; depending on the object code format, the processor type may be recorded in the object file. If it is critical that the `as` output match a specific architecture, specify that architecture explicitly.

`-b`

Add code to collect information about conditional branches taken, for later optimization using branch prediction bits. (The conditional branch instructions have branch prediction bits in the CA, CB, and CC architectures.) If *`BR`* represents a conditional branch instruction, the following represents the code generated by the assembler when `-b` is specified.

```
         call          increment  routine
         word          0          #  pre-counter
Label:   BR
  call   increment                routine
        .word          0          #  post-counter
```

The counter following a branch records the number of times that branch was *not* taken; the difference between the two counters is the number of times the branch *was* taken.

A table of every such `Label` is also generated, so that the external postprocessor `gbr960` (supplied by Intel) can locate all the counters. This table is always labeled `__BRANCH_TABLE__`; this is a local symbol to permit collecting statistics for many separate object files. The table is word aligned, and begins with a two-word header. The first word, initialized to 0, is used in maintaining linked lists of branch tables. The second word is a count of the number of entries in the table, which follow immediately: each is a word, pointing to one of the labels illustrated in the previous example.

```
*NEXT     COUNT: N    *BRLAB 1    ...       *BRLAB N
```

`_BRANCH_TABLE_` *layout*

The first word of the header is used to locate multiple branch tables, since each object file may contain one. Normally the links are maintained with a call to an initialization routine, placed at the beginning of each function in the file. The GNU C compiler generates these calls automatically when you give it a `-b` option. For further details, see the documentation of `gbr960`.

`-no-relax`

Normally, Compare-and-Branch instructions with targets that require displacements greater than 13 bits (or that have external targets) are replaced with the corresponding compare (or `chkbit`) and branch instructions. You can use the `-no-relax` option to specify that as should generate errors instead, if the target displacement is larger than 13 bits.

This option does not affect the Compare-and-Jump instructions; the code emitted for them is *always* adjusted when necessary (depending on displacement size), regardless of whether you use `-no-relax`.

i960 floating point

`as` generates IEEE floating-point numbers for the directives `.float`, `.double`, `.extended`, and `.single`.

i960 machine directives

The following documentation discusses the features pertinent to the i960 regarding the machine directives for the GNU assembler.

`.bss` *`symbol`*, *`length`*, *`align`*

Reserve *`length`* bytes in the bss section for a local *`symbol`*, aligned to the power of two specified by *`align`*. *`length`* and *`align`* must be positive absolute expressions. This directive differs from `.lcomm` only in that it permits you to specify an alignment. See “.lcomm symbol, length” on page 75.

`.extended` *`flonums`*

`.extended` expects zero or more flonums, separated by commas; for each flonum, `.extended` emits an IEEE extended-format (80-bit) floating-point number.

`.leafproc call-lab, bal-lab`

You can use the `.leafproc` directive in conjunction with the optimized `callj` instruction to enable faster calls of leaf procedures. If a procedure is known to call no other procedures, you may define an entry point that skips procedure prolog code (and that does not depend on system-supplied saved context), and declare it as the *`bal-lab`* using `.leafproc`. If the procedure also has an entry point that goes through the normal prolog, you can specify that entry point as call-lab.

A `.leafproc` declaration is meant for use in conjunction with the optimized call instruction `callj`; the directive records the data needed later to choose between converting the `callj` into a `bal` or a `call`.

`call-lab` is optional; if only one argument is present, or if the two arguments are identical, the single argument is assumed to be the bal entry point.

`.sysproc` *`name`*, *`index`*

The `.sysproc` directive defines a name for a system pro-cedure. After you define it using `.sysproc`, you can use name to refer to the system procedure identified by *`index`* when calling procedures with the optimized call instruction `callj`.

Both arguments are required; *`index`* must be between 0 and 31 (inclusive).

i960 opcodes

All Intel 960 machine instructions are supported; see “i960 command-line options” on page 150 for a discussion of selecting the instruction subset for a particular 960 architecture. Some opcodes are processed beyond simply emitting a single corresponding instruction: `callj`, and Compare-and-Branch or Compare-and- Jump instructions with target displacements larger than 13 bits.

callj

You can write `callj` to have the assembler or the linker determine the most appropriate form of subroutine call: `call`, `bal`, or `calls`. If the assembly source contains enough information—a `.leafproc` or `.sysproc` directive defining the operand—then `as` translates the `callj`; if not, it simply emits the `callj`, leaving it for the linker to resolve.

i960 Compare-and-Branch

The 960 architectures provide combined *Compare-and-Branch* instructions that permit you to store the branch target in the lower 13 bits of the instruction word itself. However, if you specify a branch target far enough away that its address won't fit in 13 bits, the assembler can either issue an error, or convert your Compare-and-Branch instruction into separate instructions to do the compare and the branch.

Whether `as` gives an error or expands the instruction depends on two choices you can make: whether you use the `-no-relax` option, and whether you use a "*Compare and Branch*" instruction or a "*Compare and Jump*" instruction. The "*Jump*" instructions are *always* expanded if necessary; the "*Branch*" instructions are expanded when necessary *unless* you specify `-no-relax`–in which case `as` gives an error instead.

The following example shows the Compare-and-Branch instructions, their "Jump" variants, and the instruction pairs into which they may expand.

Compare and		
Branch	***Jump***	***Expanded to***
`bbc`		`chkbit; bno`
`bbs`		`chkbit; bo`
`cmpibe`	`cmpije`	`cmpi; be`
`cmpibg`	`cmpijg`	`cmpi; bg`
`cmpibge`	`cmpijge`	`cmpi; bge`
`cmpibl`	`cmpijl`	`cmpi; bl`
`cmpible`	`cmpijle`	`cmpi; ble`
`cmpibno`	`cmpijno`	`cmpi; bno`
`cmpibne`	`cmpijne`	`cmpi; bne`
`cmpibo`	`cmpijo`	`cmpi; bo`
`cmpobe`	`cmpoje`	`cmpo; be`
`cmpobg`	`cmpojg`	`cmpo; bg`
`cmpobge`	`cmpojge`	`cmpo; bge`
`cmpobl`	`cmpojl`	`cmpo; bl`
`cmpoble`	`cmpojle`	`cmpo; ble`
`cmpobne`	`cmpojne`	`cmpo; bne`

20

M68K dependent features

The following documentation discusses the features pertinent to the Motorola 68000 processor regarding the GNU assembler.

M680x0 options

The Motorola 680x0 version of `as` has a few machine dependent options.

You can use the `-l` option to shorten the size of references to undefined symbols. If you do not use the `-l` option, references to undefined symbols are wide enough for a full `long` (32 bits). (Since as cannot know where these symbols end up, `as` can only allocate space for the linker to fill in later. Since as does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only one word wide (16 bits). This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols are always less than 17 bits away.

For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a `%` before any use of a register name. This is intended to let the assembler distinguish between C variables and functions named `a0` through `a7`, and so on. The `%` is always accepted, but is not required for certain configurations, notably `sun3`. The `--register-prefix-optional` option may be used to permit omitting the `%` even for configurations for which it is normally required. If this is done, it will generally be impossible to refer to C variables and functions with the same names as register names.

`as` can assemble code for several different members of the Motorola 680x0 family. The default depends upon how `as` was configured when it was built; normally, the default is to assemble code for the 68020 microprocessor. The following options may be used to change the default. These options control which instructions and addressing modes are permitted. The members of the 680x0 family are very similar. For detailed information about the differences, see the Motorola manuals.

`-m68000`
`-m68008`
`-m68302`
: Assemble for the 68000. `-m68008` and `-m68302` are synonyms for `-m68000`, since the chips are the same from the point of view of the assembler.

`-m68010`
: Assemble for the 68010.

`-m68020`
: Assemble for the 68020. This is normally the default.

`-m68030`
: Assemble for the 68030.

`-m68040`
: Assemble for the 68040.

`-m68060`
: Assemble for the 68060.

`-mcpu32`
`-m68331`
`-m68332`
`-m68333`
`-m68340`
`-m68360`
: Assemble for the CPU32 family of chips.

`-m68881`
`-m68882`
: Assemble 68881 floating point instructions. This is the default for the 68020, 68030, and the CPU32. The 68040 and 68060 always support floating point instructions.

`-mno-68881`
: Do not assemble 68881 floating point instructions. This is the default for 68000 and the 68010. The 68040 and 68060 always support floating point instructions, even if this option is used.

`-m68851`
: Assemble 68851 MMU instructions. This is the default for the 68020, 68030, and 68060. The 68040 accepts a somewhat different set of MMU instructions; `-m68851` and `-m68040` should not be used together.

`-mno-68851`
: Do not assemble 68851 MMU instructions. This is the default for the 68000, 68010, and the CPU32. The 68040 accepts a somewhat different set of MMU instructions.

This syntax for the Motorola 680x0 was developed at MIT.

The 680x0 version of `as` uses instructions names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, `movl` is equivalent to `mov.l`.

M680x0 syntax

The following documentation discusses the M680x0 syntax.

apc stands for any of the address registers (`%a0` through `%a7`), the program counter (`%pc`), the zero-address relative to the program counter (`%zpc`), a suppressed address register (`%za0` through `%za7`), or it may be omitted entirely.

The use of *size* means one of `w` or `l`, and it may be omitted, along with the leading colon, unless a scale is also specified. The use of `scale` means one of `1`, `2`, `4`, or `8`, and it may always be omitted along with the leading colon.

The following addressing modes are understood.

Immediate
: `#number`

Data Register
: `%d0` through `%d7`

Address Register
: `%a0` through `%a7`
: `%a7` is also known as `%sp`; i.e., the Stack Pointer.
: `%a6` is also known as `%fp`; i.e., the Frame Pointer.

Address Register Indirect
: `%a0@` through `%a7@`

Address Register Postincrement
: `%a0@+` through `%a7@+`

Address Register Predecrement
: `%a0@-` through `%a7@-`

Indirect Plus Offset
: `apc@(number)`

Index
: `apc@(number,register:size:scale)`
: The *number* may be omitted.

Postindex
: `apc@(number)@(onumber,register:size:scale)`
: The *onumber* or the *register*, but not both, may be omitted.

Preindex
: `apc@(number,register:size:scale)@(onumber)`
: The number may be omitted. Omitting the register produces the *Postindex* addressing mode.

Absolute

`symbol`, or `digits`, optionally followed by `:b`, `:w`, or `:l`.

Standard Motorola syntax differences

The standard Motorola syntax for this chip differs from the syntax already discussed (see “Syntax” on page 31). `as` can accept Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction.

The two kinds of syntax are fully compatible.

In the following, `apc` stands for any of the address registers (`%a0` through `%a7`), the program counter (`%pc`), the zero-address relative to the program counter (`%zpc`), or a suppressed address register (`%za0` through `%za7`). The use of *`size`* means one of `w` or `l`, and it may always be omitted along with the leading dot. The use of *`scale`* means one of `1`, `2`, `4`, or `8`, and it may always be omitted along with the leading asterisk. The following additional addressing modes are understood.

Address Register Indirect

`%a0` through `%a7`
`%a7` is also known as `%sp`; i.e., the Stack Pointer.
`%a6` is also known as `%fp`; i.e., the Frame Pointer.

Address Register Postincrement

`(%a0)+` through `(%a7)+`

Address Register Predecrement

`-(%a0)` through `-(%a7)`

Indirect Plus Offset

`number``(`*`%a0`*`)` through *`number`*`(`*`%a7`*`)`, or *`number`*`(`*`%pc`*`)`.
The number may also appear within the parentheses, as in `(`*`number`*`,`*`%a0`*`)`. When used with the *`pc`*, the *`number`* may be omitted (with an address register, omitting the *`number`* produces Address Register Indirect mode).

Index

`number``(`*`apc`*`,`*`register`*`.`*`size`*`*`*`scale`*`)`
The *`number`* may be omitted, or it may appear within the parentheses. The *`apc`* may be omitted. The *`register`* and the *`apc`* may appear in either order. If both *`apc`* and *`register`* are address registers, and the *`size`* and *`scale`* are omitted, then the first register is taken as the base register, and the second as the index register.

Postindex

`([`*`number`*`,`*`apc`*`],`*`register`*`.`*`size`*`*`*`scale`*`,`*`onumber`*`)`
The *`onumber`*, or the *`register`*, or both, may be omitted. Either the *`number`* or the *`apc`* may be omitted, but not both.

Preindex

`([`*`number`*`,`*`apc`*`,`*`register`*`.`*`size`*`*`*`scale`*`],`*`onumber`*`)`

The *`number`*, or the *`apc`*, or the *`register`*, or any two of them, may be omitted. The *`onumber`* may be omitted. The *`register`* and the *`apc`* may appear in either order. If both *`apc`* and *`register`* are address registers, and the *`size`* and scale are omitted, then the first register is taken as the base register, and the second as the index register.

M680x0 floating point

Packed decimal (P) format floating literals are not supported. Feel free to add the code!

The floating point formats generated by directives are the following.

`.float`
: `Single` precision floating point constants.

`.double`
: `Double` precision floating point constants.

`.extend`

`.ldouble`
: `Extended` precision (`long double`) floating point constants.

M680x0 machine directives

In order to be compatible with the Sun assembler, the 680x0 assembler understands the following directives.

`.data1`
: This directive is identical to a `.data 1` directive.

`.data2`
: This directive is identical to a `.data 2` directive.

`.even`
: This directive is a special case of the `.align` directive; it aligns the output to an even byte boundary.

`.skip`
: This directive is identical to a `.space` directive.

M680x0 opcodes

The following documentation discusses the features pertinent to the M680x0 regarding the opcodes for the GNU assembler.

M680x0 branch improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that reach the target. Generally these mnemonics are made by substituting `j` for `b` at the start of a Motorola mnemonic. The following summarizes the pseudo-operations.

```
           Displacement
           +---------------------------------------------------
           |                  6802068000/10
 Pseudo-Op |BYTE   WORD    LONG    LONG        non-PC relative
           +---------------------------------------------------
      jbsr |bsrs   bsr     bsrl    jsr         jsr
       jra |bras   bra     bral    jmp         jmp
 *     jXX |bXXs   bXX     bXXl    bNXs;jmpl   bNXs;jmp
 *    dbXX |dbXX   dbXX        dbXX;bra;jmpl
 *    fjXX |fbXXw  fbXXw   fbXXl               fbNXw;jmp

 XX: condition
 NX: negative of condition XX
```

`*` See the following description of the pseudo-ops with asterisks in the above table..

`jbsr`

`jra`

These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target.

`jXX`

Here, `jXX` stands for an entire family of pseudo-operations, where *XX* is a conditional branch or condition-code test. The full list of pseudo-ops in this family is the following:

```
jhi   jls   jcc   jcs   jne   jeq   jvc
jvs   jpl   jmi   jge   jlt   jgt   jle
```

For the cases of non-PC relative displacements and long displacements on the 68000 or 68010, `as` issues a longer code fragment in terms of *NX*, the opposite condition to *XX*.

```
jXX foo
```

For the non-PC relative case in the previous example , `as` gives the following output.

```
bNXs oof
jmp foo
oof:
```

`dbXX`

The full family of pseudo-operations covered here is the following.

```
dbhi   dbls   dbcc   dbcs   dbne   dbeq   dbvc
dbvs   dbpl   dbmi   dbge   dblt   dbgt   dble
dbf    dbra   dbt
```

Other than for word and byte displacements, when the source reads `dbXX foo`, **`as`** emits the following output.

```
         dbXX oo1
         bra oo2
  oo1:  jmpl foo
  oo2:
```

`fjXX`

This family includes the following.

```
fjne    fjeq    fjge    fjlt    fjgt    fjle     fjf
fjt     fjgl    fjgle   fjnge   fjngl   fjngle   fjngt
fjnle   fjnlt   fjoge   fjogl   fjogt   fjole    fjolt
fjor    fjseq   fjsf    fjsne   fjst    fjueq    fjuge
fjugt   fjule   fjult   fjun
```

For branch targets that are not PC relative, `as` emits the following output when it encounters `fjXX foo`.

```
      fbNX oof
      jmp foo
oof:
```

M680x0 special characters

The immediate character is `#` for Sun compatibility. The line-comment character is `|`. If a `#` appears at the beginning of a line, it is treated as a comment unless it looks like `# line file`, in which case it is treated normally.

21

MIPS dependent features

The following documentation discusses the features pertinent to the MIPS processors regarding the GNU assembler.

MIPS processors that have support are the MIPS R2000, R3000, R4000 and R6000. For information about the MIPS instruction set, see ***MIPS RISC Architecture***, by Kane and Heindrich (Prentice-Hall). For an overview of MIPS assembly conventions, see "Appendix D: Assembly Language Programming" in the same work.

Assembler options for MIPS

The MIPS configurations of GNU `as` support the following special options:

`-G` *`num`*
: This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format. The default value is 8.

`-EB`
`-EL`
: Any MIPS configuration of `as` can select big-endian or little-endian output at run time (unlike the other GNU development tools, which must be configured for one or the other). Use `-EB` to select big-endian output, and `-EL` for little-endian.

`-mips1`
`-mips2`
`-mips3`
: Generate code for a particular MIPS Instruction Set Architecture level. `-mips1` corresponds to the R2000 and R3000 processors, `-mips2` to the R6000 processor, and `-mips3` to the R4000 processor. You can also switch instruction sets during the assembly; see “Directives to override the ISA level for MIPS” on page 169.

`-m4650`
`-no-m4650`
: Generate code for the MIPS R4650 chip. This tells the assembler to accept the `mad` and `madu` instruction, and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4650` turns off this option.

`-m4010`
`-no-m4010`
: Generate code for the LSI R4010 chip. This tells the assembler to accept the R4010 specific instructions (`addciu`, `ffc`, etc.), and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4010` turns off this option.

`-mcpu=CPU`
: Generate code for a particular MIPS CPU. This has little effect on the assembler, but it is passed by `gcc`.

`-nocpp`
: This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With GNU `as`, there is no need for `-nocpp`, because the GNU assembler itself never runs the C preprocessor.

`--trap`
`--no-break`
: `as` automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes as to generate code to take a trap exception rather than a break exception when an error is detected. The trap instructions are only supported at Instruction Set Architecture level 2 and higher.

`--break`
`--no-trap`
: Generate code to take a break exception rather than a trap exception when an error is detected. This is the default.

MIPS ECOFF object code

Assembling for a MIPS ECOFF target supports some additional sections besides the usual `.text`, `.data` and `.bss`. The additional sections are `.rdata`, used for read-only data, `.sdata`, used for small data, and `.sbss`, used for small common objects.

When assembling for ECOFF, the assembler uses the `$gp ($28)` register to form the address of a *small object*. Any object in the `.sdata` or `.sbss` sections is considered "small" in this sense. For external objects, or for objects in the `.bss` section, you can use the `gcc` '`-G`' option to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller uses `$gp`.

Passing '`-G 0`' to `as` prevents it from using the `$gp` register on the basis of object size (but the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case). The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it. The size of an external object may be set with the `.extern` directive. For example, `.extern sym,4` declares that the object at `sym` is 4 bytes in length, while leaving `sym` otherwise undefined.

Using small ECOFF objects requires linker support, and assumes that the `$gp` register is correctly initialized (normally done automatically by the startup code). MIPS ECOFF assembly code must not modify the `$gp` register.

Directives for debugging information for MIPS

MIPS ECOFF `as` supports several directives used for generating debugging information which are not support by traditional MIPS assemblers. These are `.def`, `.endef`, `.dim`, `.file`, `.scl`, `.size`, `.tag`, `.type`, `.val`, `.stabd`, `.stabn`, and `.stabs`. The debugging information generated by the three `.stab` directives can only be read by GDB, not by traditional MIPS debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers!

Directives to override the ISA level for MIPS

GNU `as` supports an additional directive to change the MIPS Instruction Set Architecture level on the fly: `.set mips`*n*. *n* should be a number from 0 to 3. A value from 1 to 3 makes the assembler accept instructions for the corresponding isa level, from that point on in the assembly. `.set mips`*n* affects not only which instructions are permitted, but also how certain macros are expanded. `.set mips0` restores the ISA level to its original level: either the level you selected with command line options, or the default for your configuration. You can use this feature to permit specific R4000 instructions while assembling in 32-bit mode. Use this directive with care! Traditional MIPS assemblers do not support this directive.

Directives for extending MIPS 16 bit instructions

By default, MIPS 16 instructions are automatically extended to 32 bits when necessary. The directive ‘`.set noautoextend`’ will turn this off. When ‘`.set noautoextend`’ is in effect, any 32 bit instruction must be explicitly extended with the ‘.e’ modifier (e.g., ‘`li.e $4,1000`’). The directive ‘`.set autoextend`’ may be used to once again automatically extend instructions when necessary.

This directive is only meaningful when in MIPS 16 mode. Traditional MIPS assemblers do not support this directive.

Directive to mark data as an instruction for Mfor MIPSIPS

The `.insn` directive tells **as** that the following data is actually instructions. This makes a difference in MIPS 16 mode: when loading the address of a label which precedes instructions, `as` automatically adds 1 to the value, so that jumping to the loaded address will do the right thing.

Directives to save and restore options for MIPS

The directives, `.set push` and `.set pop`, may be used to save and restore the current settings for all the options which are controlled by .set. The `.set push` directive saves the current settings on a stack. The `.set pop` directive pops the stack and restores the settings.

These directives can be useful inside an macro which must change an option such as the ISA level or instruction reordering but does not want to change the state of the code which invoked the macro.

Traditional MIPS assemblers do not support these directives.

22

SPARC dependent features

The following documentation discusses the features pertinent to the SPARCprocessors regarding the GNU assembler.

SPARC options

The SPARC chip family includes several successive levels (or other variants) of chip, using the same core instruction set, but including a few additional instructions at each level.

By default, `as` assumes the core instruction set (SPARC v6), but “bumps” the architecture level as needed: it switches to successively higher architectures as it encounters instructions that only exist in the higher levels.

If not configured for SPARC v9 (`sparc64-*-*`) `as` will not bump passed `sparclite` by default, an option must be passed to enable the v9 instructions.

`as` treats `sparclite` as being compatible with v8, unless an architecture is explicitly requested. SPARC v9 is always incompatible with `sparclite`.

`-Av6 | -Av7 | -Av8 | -Asparclite | -Av9 | -Av9a`
: Use one of the `-A` options to select one of the SPARC architectures explicitly. If you select an architecture explicitly, `as` reports a fatal error if it encounters an instruction or feature requiring a higher level.

`-xarch=v8plus | -xarch=v8plusa`
: For compatibility with the Solaris v9 assembler. These options are equivalent to `-Av9` and `-Av9a`, respectively.

`-bump`
: Warn whenever it is necessary to switch to another level. If an architecture level is explicitly requested, `as` will not issue warnings until that level is reached, and will then bump the level as required (except between incompatible levels).

SPARC floating point

The SPARC uses IEEE floating-point numbers.

SPARC machine directives

The SPARC version of `as` supports the following additional machine directives:

`.align`
: This must be followed by the desired alignment in bytes.

`.common`
: This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.comm`, but the syntax is different.

`.half`
: This is functionally identical to `.short`.

`.proc`
: This directive is ignored. Any text following it on the same line is also ignored.

`.reserve`
: This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.lcomm`, but the syntax is different.

`.seg`
: This must be followed by `"text"`, `"data"`, or `"data1"`. It behaves like `.text`, `.data`, or `.data 1`.

`.skip`
: This is functionally identical to the `.space` directive.

`.word`
: On the SPARC, the `.word` directive produces 32 bit values, instead of the 16 bit values it produces on many other machines.

`.xword`
: On the SPARC V9 processor, the `.xword` directive produces 64 bit values.

23

Vax dependent features

The following documentation discusses the features pertinent to the Vax processors regarding the GNU assembler.

Vax command-Line options

The Vax version of `as` accepts any of the following options, gives a warning message that the option was ignored and proceeds. These options are for compatibility with scripts designed for other people's assemblers.

`-D (Debug)`
`-S (Symbol Table)`
`-T (Token Trace)`
: These are obsolete options used to debug old assemblers.

`-d (Displacement size for JUMPs)`
: This option expects a number following the `-d`. Like options that expect filenames, the number may immediately follow the `-d` (old standard) or constitute the whole of the command line argument that follows `-d` (GNU standard).

`-V (Virtualize Interpass Temporary File)`
: Some other assemblers use a temporary file. This option commanded them to keep the information in active memory rather than in a disk file. `as` always does this, so this option is redundant.

`-J (JUMPify Longer Branches)`
: Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Some other assemblers would emit short and medium branches, unless told by this option to emit short and long branches.

`-t (Temporary File Directory)`
: Some other assemblers may use a temporary file, and this option takes a filename being the directory to site the temporary file. Since `as` does not use a temporary disk file, this option makes no difference. `-t` needs exactly one filename.

The Vax version of the assembler accepts two options when compiled for VMS. They are `-h`, and `-+`. The `-h` option prevents `as` from modifying the symbol-table entries for symbols that contain lowercase characters (I think). The `-+` option causes `as` to print warning messages if the `FILENAME` part of the object file, or any symbol name is larger than 31 characters.

The `-+` option also inserts some code following the `_main` symbol so that the object file is compatible with Vax-11 "C".

Vax floating point

Conversion of flonums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit. `D`, `F`, `G` and `H` floating point formats are understood. Immediate floating literals (e.g., `S'$6.9`) are rendered correctly. Again, rounding is towards zero in the boundary case. The `.float` directive produces `f` format numbers. The `.double` directive produces `d` format numbers.

Vax machine directives

The Vax version of the assembler supports four directives for generating Vax floating point constants. They are described in the following.

`.dfloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `d` format 64-bit floating point constants.

`.ffloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `f` format 32-bit floating point constants.

`.gfloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `g` format 64-bit floating point constants.

`.hfloat`
: This expects zero or more flonums, separated by commas, and assembles Vax `h` format 128-bit floating point constants.

Vax opcodes

All DEC mnemonics are supported. Beware that `case...` instructions have exactly 3 operands. The dispatch table that follows the `case...` instruction should be made with `.word` statements. As far as we know, this is compatible with all Unix assemblers.

Vax branch improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that reaches the target. Generally these mnemonics are made by substituting `j` for `b` at the start of a DEC mnemonic.

This feature is included both for compatibility and to help compilers. If you do not need this feature, avoid these opcodes. What follows are the mnemonics, and the code into which they can expand.

`jbsb`
: `Jsb` is already an instruction mnemonic, so we chose `jbsb`.

  (byte displacement)

  `bsbb``...`

  (word displacement)

  `bsbw``...`

  (long displacement)

jsb...

`jbr`
`jr`
Unconditional branch.

(byte displacement)
brb...

(word displacement)
brw...

(long displacement)
jmp...

jCOND
COND may be any one of the conditional branches `neq`, `nequ`, `eql`, `eqlu`, `gtr`, `geq`, `lss`, `gtru`, `lequ`, `vc`, `vs`, `gequ`, `cc`, `lssu`, `cs`. *COND* may also be one of the bit tests `bs`, `bc`, `bss`, `bcs`, `bsc`, `bcc`, `bssi`, `bcci`, `lbs`, `lbc`. *NOTCOND* is the opposite condition to *COND*.

(byte displacement)
bCOND...

(word displacement)
bNOTCOND foo ; brw...; foo:

(long displacement)
bNOTCOND foo ; jmp...; foo:

`jacb`*X*
X may be one of `b d f g h l w`.

(word displacement)
OPCODE...

(long displacement)
```
OPCODE..., foo ;
  brb bar ; foo: jmp... ;
  bar:
```

`jaob`*YYY*
YYY may be one of `lss leq`.

`jsob`*ZZZ*
ZZZ may be one of `geq gtr`.

(byte displacement)
OPCODE...

(word displacement)
```
OPCODE... , foo ;
  brb bar ;
  foo: brw destination ;
  bar:
```

(long displacement)

```
OPCODE..., foo ;
 brb bar ;
 foo: jmp destination ;
 bar:
```

`aobleq`
`aoblss`
`sobgeq`
`sobgtr`

(byte displacement)

```
OPCODE...
```

(word displacement)

```
OPCODE... , foo ;
 brb bar ;
 foo: brw destination ;
 bar:
```

(long displacement)

```
OPCODE..., foo ;
 brb bar ;
 foo: jmp destination ;
 bar:
```

Vax operands

The immediate character is `$` for Unix compatibility, not `#` as DEC writes it.

The indirect character is `*` for Unix compatibility, not `@` as DEC writes it.

The displacement sizing character is `‘` (an accent grave) for Unix compatibility, not `^` (a circumflex) as DEC writes it. The letter preceding `‘` may have either case. `G` is not understood, but all other letters (`b`, `i`, `l`, `s`, `w`) are understood.

Register names understood are `r0 r1 r2...r15 ap fp sp pc`. Upper and lower case letters are equivalent. For instance

```
tstb *w‘$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

Unsupported Vax assembler properties

Vax bit fields can not be assembled with `as`. Someone can add the required code if they really need it.

24

Acknowledgments

The following individuals contributed to the develoment of the GNU assembler.

- Dean Elsner wrote the original GNU assembler for the VAX.
- Jay Fenlason maintained GAS for a while, adding support for GDB-specific debug information and the 68k series machines, most of the pre-processing pass, and extensive changes in `messages.c`, `input-file.c`, `write.c`.
- K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file format back ends (including heavy rewrite, testing, an integration of the COFF and `b.out` back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ANSI C including full prototypes, added support for m680[34]0 and cpu32, did considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, RS6000, and HP300/HPUX host ports, updated "`know`" assertions and made them work, much other reorganization, cleanup, and lint.
- Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.
- The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

- The Intel 80386 machine description was written by Eliot Dresselhaus.
- Minh Tran-Le at IntelliCorp contributed some AIX 386 support.
- The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish In-stitute of Computer Science.
- Keith Knowles at the Open Software Foundation wrote the original MIPS back end (`tc-mips.c`, `tc-mips.h`), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support `a.out` format.
- Support for the Zilog Z8k and Hitachi H8/300 and H8/500 processors (tc-z8k, tc-h8300, tc-h8500), and IEEE 695 object file format (`obj-ieee`), was written by Steve Chamberlain of Cygnus. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.
- John Gilmore built the AMD 29000 support, added `.include` support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g., `jsr`), while synthetic instructions remained shrinkable (`jbsr`). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.
- Ian Lance Taylor of Cygnus merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS, ECOFF and ELF targets, and made a few other minor patches.
- Steve Chamberlain made `as` able to generate listings.
- Hewlett-Packard contributed support for the HP9000/300.
- Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus.
- Support for ELF format files has been worked on by Mark Eichin of Cygnus (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn of Cygnus (SPARC, and some initial 64-bit support).

- Several engineers at Cygnus have also provided many small bug fixes and configuration enhancements. Many others have contributed large or small bugfixes and enhancements.

If you have contributed significant work and are not mentioned on this list, and want to be, let Cygnus know. Some of the history has been lost; Cygnus is not intentionally neglecting anyone's contributions.

Send mail to the maintainer, and we'll correct the situation. Currently, the maintainer is Ken Raeburn (email address: `raeburn@cygnus.com`).

GNUPRO™ TOOLKIT

Using LD

July, 1998

98r1

CYGNUS

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

1

Overview of the GNU linker, `ld`

The GNU linker, `ld`, combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run `ld`.

The following documentation discusses using the GNU linker.

- "Invocation of ld" on page 191
- "Linker scripts" on page 205
- "Machine dependent features" on page 241
- "BFD" on page 245
- "MRI compatible script files for ld" on page 251

`ld` accepts Linker Command Language files written in a superset of AT&T's *Link Editor Command Language* syntax, to provide explicit and total control over the linking process.

This version of `ld` uses the general purpose BFD libraries to operate on object files. This allows `ld` to read, combine, and write object files in many different formats—for example, COFF or `a.out`. Different formats may be linked together to produce any available kind of object file. See "BFD" on page 245 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` 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. The following documentation discusses using the GNU linker with such choices.

- “Command line options for ld” on page 192
- “Environment variables for ld” on page 204

See also “Linker scripts” on page 205.

Command line options for ld

The following example summarizes the options you can use on the ld command line.

```
ld [ -o output ] objfile : ::
   [ -Aarchitecture ][ -b input-format ]
   [ -Bstatic ] [ -Bdynamic ] [ -Bsymbolic ]
   [ -c MRI-commandfile ] [ -d | -dc | -dp ]
   [ -defsym symbol=expression ]
   [ -dynamic-linker file ] [ -embedded-relocs ]
   [ -e entry ] [ -F ] [ -F format ]
   [ -format input-format ] [ -g ] [ -G size ]
   [ -help ] [ -i ] [ -l archive ] [-Lsearchdir ]
   [ -M ] [ -Map mapfile ][-m emulation ]
   [ -N | -n ] [ -noinhibit-exec ] [ -no-keep-memory ]
   [ -oformat output-format ][-R filename ]
   [ -relax ] [ -retain-symbols-file filename ]
   [ -r | -Ur ] [ -rpath dir ] [ -rpath-link dir ]
   [ -S ] [ -s ] [ -soname name ] [ -shared ]
   [ -sort-common ] [ -stats ] [ -T commandfile ]
   [ -Ttext org ] [ -Tdata org ]
   [ -Tbss org ] [ -t ] [ -traditional-format ]
   [ -u symbol] [ -V] [ -v] [ -verbose] [ -version ]
   [ -warn-common ] [ -warn-constructors] [ -warn-once ]
   [ -y symbol ] [ -X ] [-x ]
   [ -( [ archives ] -) ]
   [ --start-group [ archives ] --end-group ]
   [ -split-by-reloc count ] [ -split-by-file ]
   [ --whole-archive ]
```

This plethora of command-line options may seem intimidating, but in actual practice few of them are used in any particular context. For instance, a frequent use of ld is to link standard Unix object files on a standard, supported Unix system. On such a system, to link a file, hello.o, 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 file /lib/crt0.o with hello.o and the library libc.a, which will come from the standard search directories. (See the discussion of search directories with the -L option.)

The command-line options to ld may be specified in any order, and may be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that specific option.

The exceptions—which may meaningfully be used more than once—are the following: `-A`, `-b` (or its synonym, `-format`), `-defsym`, `-L`, `-l`, `-R`, `-u`, and `-(` (or its synonym, `--start-group`).

The list of object files to be linked together, shown as *`objfile`*`...`, may follow, precede, or be mixed in with command-line options, except that an *`objfile`* argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but you can specify other forms of binary input files using '`-l`', '`-R`', and the script command language. If no binary input files at all are specified, the linker does not produce any output, and issues the message '`No input files`'.

If the linker can not recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link (either the default linker script or the one specified by using '`-T`'). This feature permits the linker to link against a file which appears to be an object or an archive, but actually merely defines some symbol values, or uses `INPUT` or `GROUP` to load other objects.

For options whose names are a single letter, option arguments must either follow the option letter without intervening whitespace, or be given as separate arguments immediately following the option that requires them.

For options whose names are multiple letters, either one dash or two can precede the option name; for example, `--oformat` and `-oformat` are equivalent. Arguments to multiple-letter options must either be separated from the option name by an equals sign, or be given as separate arguments immediately following the option that requires them. For example, --`oformat srec` and `--oformat=srec` are equivalent. Unique abbreviations of the names of multiple-letter options are accepted.

`-A`*`architecture`*

In the current release of `ld`, this option is useful only for the Intel 960 family of architectures. In that `ld` configuration, the *`architecture`* argument identifies the particular architecture in the 960 family, enabling some safeguards and modifying the archive-library search path. See "ld and the Intel 960 processors" on page 243 for details. Future releases of `ld` may support similar functionality for other architecture families.

`-b` *`input-format`*

`ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`-b`' option to specify the binary format for input object files that follow this option on the command line. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be 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 '`objdump -i`'.) '`-format` *`input-format`*' has the same effect, as does the script command `TARGET`. See "BFD" on page 245.

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 "Environment variables for ld" on page 204. You can also define the input format from a script, using the command `TARGET`.

`-Bstatic`
: Do not link against shared libraries. This is only meaningful on platforms for which shared libraries are supported.

`-Bdynamic`
: 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.

`-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.

`-c` *`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 ld" on page 251. Introduce MRI script files with the option, '`-c`'; use the '`-T`' option to run linker scripts written in the general-purpose `ld` scripting language. If *`MRI-cmdfile`* does not exist, `ld` looks for it in the directories specified by any '`-L`' options.

`-d`
`-dc`
`-dp`
: These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with '`-r`'). The script command, `FORCE_COMMON_ALLOCATION`, has the same effect.

`-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 "Simple assignments" on page 212).

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

`-dynamic-linker` *`file`*
Set the name of the dynamic linker. This is only meaningful when generating dynamically linked ELF executables. The default dynamic linker is normally correct; don't use this unless you know what you are doing.

`-embedded-relocs`
This option is only meaningful when linking MIPS embedded PIC code, generated by the `-membedded-pic` option to the GNU compiler and assembler. It causes the linker to create a table which may be used at runtime to relocate any data which was statically initialized to pointer values. See the code in `testsuite/ld-empic` for details.

`-e` *`entry`*
Use *`entry`* as the explicit symbol for beginning execution of your program, rather than the default entry point. See "Setting the entry point" on page 209 for a discussion of defaults and other ways of specifying the entry point.

`-F`
`-F` *`format`*
Ignored. Some older linkers used this option throughout a compilation toolchain for specifying object-file format for both input and output object files.

The mechanisms `ld` uses for this purpose (the '`-b`' or '`-format`' options for input files, '`-oformat`' option or the `TARGET` command in linker scripts for output files, the `GNUTARGET` environment variable) are more flexible, but `ld` accepts the '`-F`' option for compatibility with scripts written to call the old linker.

`-format` *`input-format`*
Synonym for '`-b` *`input-format`*'.

`-g`
Ignored. Provided for compatibility with other tools.

`-G`*`value`*
`-G` *`value`*
Set the maximum size of objects to be optimized using the GP register to *`size`* under MIPS ECOFF. Ignored for other object file formats.

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

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

`-l`*`ar`*
: Add archive file *`archive`* to the list of files to link. This option may be used any number of times. `ld` will search its path-list for occurrences of `lib`*`ar`*`.a` for every *`archive`* specified.

`-L`*`searchdir`*

`-L` *`searchdir`*
: Add path, *`searchdir`*, to the list of paths that `ld` will search for archive libraries and `ld` control scripts. You may use this option any number of times. The 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 "Environment variables for ld" on page 204. 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.

`-M`
: Print (to the standard output) a link map—diagnostic information about where symbols are mapped by `ld`, and information on global common storage allocation.

`-Map` *`mapfile`*
: Print to the file *`mapfile`* a link map—diagnostic information about where symbols are mapped by `ld`, and information on global common storage allocation.

`-m`*`emulation`*

`-m` *`emulation`*
: Emulate the *`emulation`* linker. You can list the available emulations with the `--verbose` or `-V` options. The default depends on `ld`'s configuration.

`-N`
: Set the text and data sections to be readable and writable. Also, do not page-align the data segment. If the output format supports Unix style magic numbers, mark the output as `OMAGIC`.

`-n`
: Set the text segment to be read only, and mark the output as `NMAGIC` if possible.

`-noinhibit-exec`
: Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever.

`-no-keep-memory`
: `ld` normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This 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.

`-o` *`output`*
: Use *`output`* as the name for the program produced by `ld`; if this option is not specified, the name `a.out` is used by default. The script command `OUTPUT` can also specify the output file name.

`-oformat` *`output-format`*
: `ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`-oformat`' option to specify the binary format for the output object file. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be configured to produce as a default output format the most usual format on each machine. *`output-format`* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it. See "BFD" on page 245.

`-R` *`filename`*
: Read symbol names and their addresses from *`filename`*, but do not relocate it or include it in the output. This allows your output file to refer symbolically to absolute locations of memory defined in other programs.

`-relax`
: An option with machine dependent effects. Currently this option is only supported on the H8/300 and the Intel 960. See "ld and the H8/300 processors" on page 242 and "ld and the Intel 960 processors" on page 243.

  On some platforms, the '`-relax`' option performs global optimizations that become possible when the linker resolves addressing in the program, such as relaxing address modes and synthesizing new instructions in the output object file.

  On platforms where this is not supported, '`-relax`' is accepted, but ignored.

`-retain-symbols-file` *`filename`*
: Retain only the symbols listed in the file *`filename`*, discarding all others. *`filename`* is simply a flat file, with one symbol name per line. This option is especially useful in environments (such as VxWorks) where a large global symbol table is accumulated gradually, to conserve 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

2: Invocation of `ld`

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 patch 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.

-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. The difference between -rpath and -rpath-link is that directories specified by -rpath are included in the executable to use at runtime, whereas 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.

`-r`
: Generate relocatable output—i.e., generate an output file that can in turn serve as input to `ld`. This is often called partial linking. As a side effect, in environments that support standard Unix magic numbers, this option also sets the output file's magic number to `OMAGIC`. If this option is not specified, an absolute file is produced. When linking C++ programs, this option will not resolve references to constructors; to do that, use '`-Ur`'.

  This option does the same thing as '`-i`'.

`-S`
: Omit debugger symbol information (but not all symbols) from the output file.

`-s`
: Omit all symbol information from the output file.

`-soname` *`name`*
: When creating an ELF shared object, set the internal `DT SONAME` field to the specified name. When an executable is linked with a shared object which has a `DT SONAME` field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the `DT SONAME` field rather than the using the file name given to the linker.

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

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

`-stats`

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

`-Tbss` *`org`*
`-Tdata` *`org`*
`-Ttext` *`org`*

Use *`org`* as the starting address for—respectively—the `bss`, `data`, or the `text` segment of the output file. *`org`* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '`0x`' usually associated with hexadecimal values.

`-T`*`commandfile`*
`-T` *`commandfile`*

Read link commands from the file, *`commandfile`*. These commands replace `ld`'s default link script (rather than adding to it), so *`commandfile`* must specify everything necessary to describe the target format. If *`commandfile`* does not exist, `ld` looks for it in the directories specified by any preceding '`-L`' options. Multiple '`-T`' options accumulate.

`-t`

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

`-traditional-format`

For some targets, the output of `ld` is different in some ways from the output of some existing linker. This switch requests `ld` to use the traditional format instead. For example, on SunOS, `ld` combines duplicate entries in the symbol string table. This can reduce the size of an output file with full debugging information by over 30 percent. Unfortunately, the SunOS `dbx` program can not read the resulting program (`gdb` has no trouble). The '`-traditional-format`' switch tells `ld` to not combine duplicate entries.

`-u` *`symbol`*

Force *`symbol`* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. '`-u`' may be repeated with different option arguments to enter additional undefined symbols.

`-Ur`

For anything other than C++ programs, this option is equivalent to '`-r`': it generates relocatable output—i.e., an output file that can in turn serve as input to `ld`. When linking C++ programs, '`-Ur`' does resolve references to constructors, unlike '`-r`'. It does not work to use '`-Ur`' on files that were themselves linked with '`-Ur`'; once the constructor table has been built, it cannot be added to. Use '`-Ur`' only for the last partial link, and '`-r`' for the others.

`--verbose`
Display the version number for `ld` and list the linker emulations supported. Display which input files can and cannot be opened.

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

`-version`
Display the version number for `ld` and exit.

`-warn-common`
Warn when a common symbol is combined with another common symbol or with a symbol definition. Unix linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get some warnings about symbols in the libraries as well as in your programs.

There are three kinds of global symbols, illustrated here by C examples:

'`int i = 1;`'
A definition, which goes in the initialized data section of the output file.

'`extern int i;`'
An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.

'`int i;`'
A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file. The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.

The '`-warn-common`' option can produce 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-once`

Only warn once for each undefined symbol, rather than once per module which refers to it.

For each archive mentioned on the command line, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library.

`-X`

Delete all temporary local symbols. For most targets, this is all local symbols whose names begin with ‘`L`’.

`-x`

Delete all local symbols.

`-y symbol`

Print the name of each linked file in which symbol appears. This option may be given any number of times. On many systems it is necessary to prepend an underscore. This option is useful when you have an undefined symbol in your link but don’t know where the reference is coming from.

`-(`*`archives`*`-)`
`--start-group` *`archives`*`--end-group`

The *`archives`* should be a list of archive files. They may be either explicit file names, or '`-l`' options.

The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved.

Using this option has a significant performance cost. Use it only when there are unavoidable circular references between two or more archives.

Environment variables for `ld`

You can change the behavior of `ld` with the environment variable `GNUTARGET`.

`GNUTARGET` determines the input-file object format if you don't use `-b` (or its synonym, '`-format`'). Its value should be one of the BFD names for an input format (see "BFD" on page 245). If there is no `GNUTARGET` in the environment, `ld` uses the natural format of the target. If `GNUTARGET` is set to `default` , then BFD attempts to discover the input format by examining binary input files; this method often succeeds, but there are potential ambiguities, since there is no method of ensuring that the magic number used to specify object-file formats is unique. However, the configuration procedure for BFD on each system places the conventional format for that system first in the search-list, so ambiguities are resolved in favor of convention.

3

Linker scripts

A *linker script* controls every link. Such a script is written in the linker command language. The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control 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 linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that is compiled 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. If the linker opens a file, which it can not recognize as an object file or as an archive file, it will try to read it 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 the '`INPUT`', '`GROUP`', or '`VERSION`' commands.

Basic linker script concepts

We need to define some basic concepts and vocabulary in order to describe the linker script language.

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 we will also call it an object file. Each object file has, among other things, a list of *sections*. We sometimes refer to a section in an input file 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. Here we will describe a simple use of it. Let’s assume your program consists only of code, initialized data, and uninitialized data. These will be in the ‘.text’, ‘.data’, and ‘.bss’ sections, respectively. Let’s assume further that these are the only sections, which appear in your input files.

For this example, let's say 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 ‘.’, 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’s it! That’s a simple and complete linker script.

Simple linker script commands

In the following documentation, the discussion describes the simple linker script commands. See also "Command line options for ld" on page 192 and "BFD" on page 245.

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 "Command line options for ld" on page 192 and "BFD" on page 245.

`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't 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 (-l`*`file`*`)`', `ld` will transform the name to

'`lib`*`file`*`.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 "Command line options for ld" on page 192. 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 "Command line options for ld" on page 192 and "BFD" on page 245.

`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 "Command line options for ld" on page 192 and "BFD" on page 245.

`FORCE_COMMON_ALLOCATION`

This command has the same effect as the '`-d`' command-line option: to make '`ld`' assign space to common symbols even if a relocatable output file is specified ('`-r`').

`NOCROSSREFS(`*section section* `...)`

This command may be used to tell '`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.

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 234.
- 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 234.
- 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 command

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 previousexample, 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 format of the ‘SECTIONS’ command is:

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

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

- An ‘ENTRY’ command (see “Setting the entry point” on page 209.)
- A symbol assignment (see “Simple assignments” on page 212)
- An output section description
- An overlay description

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 214 and “Overlay description” on page 224.

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 212)
- An input section description (see “Input section description” on page 216)
- Data values to include directly (see “Output section data” on page 219)
- A special output section keyword (see “Output section keywords” on page 220)

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 221.

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 234. 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 do something like the following declaration:

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

This works because ‘`ALIGN`’ returns the current location counter aligned upward to the specified value.

Specifying ‘*`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, which we describe; see “Input section wildcard patterns” on page 217. 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)
```

Here 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 by 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) }
```

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.

This 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 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 234. 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.

This 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 223); 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:

```
.foo { *(.foo) }
```

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 which are assigned to an output section named ‘`/DISCARD/`’ are not included in the output file.

Output section attributes

We showed above that the full description of an output section looked like this:

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

We’ve already described ‘*section*’, ‘*address*’, and ‘*output-sections-command*’. In the following discussion, we will describe the remaining section attributes.

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 206. 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, to copy the initialized data from the ROM image to its runtime address. Notice how 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`'. Here is a simple example:

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

Output section phdr

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 228.

Here is a simple example:

```
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 234. 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 219.

Here is a simple example:

```
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 as follows:

```
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 (‘*secname1*’ and ‘*secname2*’ above). 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 214.

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 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.

Here is an example. 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.

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

Note that the 'OVERLAY' command is just syntactic sugar, since everything it does can be done using the more basic commands. The above 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:

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

The '*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 214.

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`
: nitialized 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 keyword ‘`ORIGIN`’ 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 keyword ‘`LENGTH`’ may be abbreviated to ‘`len`’ or ‘`l`’.

In the following example, we specify that 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
  }
```

If you have defined a memory region named ‘`mem`’, you can direct the linker to place specific output sections into that memory region by using the ‘`>`*`region`*’ output section attribute. 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 223.

PHDRS command

The ELF object file format uses *program headers*, also knows 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`'.

This is 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 phdr" on page 223.

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.

The ‘*`type`*’ may be one of the following. The numbers indicate 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)`
: 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`
: 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 222.

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.

Here is an example of ‘`PHDRS`’. This shows 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 is simply

```
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 a few examples.

```
VERS_1.1 {
  global:
  foo1;
  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_foo1,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_foo1', 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 (i.e. '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.

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

Precedence	*Association*	*Operators*	*Notes*
(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 212.

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. For example, a script, like the following, will cause the error message '**`non constant expression for initial address`**':

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

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 a number of 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 234.

`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) & ~(exp - 1)
```

'`ALIGN`' doesn't 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 215.

`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 228. In the following example, ‘symbol_1’ and ‘symbol_2’ are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
...
```

4

Machine dependent features

The following documentation describes some machine independent features for the GNU linker for the H8/300 and the Intel 960 processors. See the following documentation for details.

- Machines with ld having no additional functionality have no documentation.
- `ld` and the Intel 960 processors

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 the Intel 960 processors

You can use the ‘-A*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).

4: Machine dependent features

5

BFD

The linker accesses object and archive files using the BFD libraries.

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

- "How it works: an outline of BFD" on page 247
- "Information loss" on page 248
- "The BFD canonical object-file format" on page 249

BFD libraries allow the linker to use the same routines to operate on object files whatever the object file format. A different object file format can be supported simply by creating a new BFD back end and adding it to the library. To conserve runtime memory, however, the linker and associated tools are usually configured to support only a subset of the object file formats available. To list all the formats available for your configuration, use `objdump -i` (see "**`objdump`**" in *The GNU Binary Utilities*).

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 248.

How it works: an outline of BFD

When an object file is opened, BFD subroutines automatically determine the format of the input object file. They then build a descriptor in memory with pointers to routines that will be used to access elements of the object file’s data structures.

As different information from the 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, that stored by the canonical format, and that needed by the destination format. A brief description of the canonical form may help you understand which kinds of data you can count on preserving across conversions.

files

Information stored on a per-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 that big- and little-endian object files may be used with one another.

sections

Each section in the input file contains the name of the section, the section's original address in the object file, size and alignment information, various flags, and pointers into other BFD data structures.

symbols

Each symbol contains a pointer to the information for the object file which originally defined it, its name, its value, and various flag bits. When a BFD back end reads in a symbol table, it relocates all symbols to make them relative to the base of the section where they were defined.

Doing this ensures that each symbol points to its containing section. Each symbol also has a varying amount of hidden private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. `ld` can operate on a collection of symbols of wildly different formats without problems.

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out`, type information is stored in the symbol table as long symbol names.

This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away.

There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example, COFF, IEEE, Oasys) and the type is simple enough to fit within one word (nearly everything but aggregates), the information will be preserved.

relocation level

Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in, and a pointer to a relocation type descriptor. Relocation is performed by passing messages through the relocation type descriptor and the symbol pointer. Therefore, relocations can be performed on output data using a relocation method that is only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation may be performed on a byte being written to a 68k COFF file, even though 68k COFF has no such relocation type.

line numbers

Object formats can contain, for debugging purposes, some form of mapping between symbols, source line numbers, and addresses in the output file. These addresses have to be relocated along with the symbol information. Each symbol with an associated list of line number records points to the first record of the list. The head of a line number list consists of a pointer to the symbol, which allows finding out the address of the function whose line number is being described. The rest of the list is made up of pairs: offsets into the section and line numbers. Any format which can simply derive this information can pass it successfully between formats (COFF, IEEE and Oasys).

6

MRI compatible script files for `ld`

To aid users making the transition to GNU `ld` from the MRI linker, `ld` can use MRI compatible linker scripts as an alternative to the more general-purpose linker scripting language described in “Command line options for ld” on page 192. The following documentation describes the use of the scripts.

MRI compatible linker scripts have a much simpler command set than the scripting language otherwise used with `ld`. GNU `ld` supports the most commonly used MRI linker commands; these commands are described in the following.

In general, MRI scripts aren’t of much use with the `a.out` object file format, since it only has three sections and MRI scripts lack some features to make use of them.

You can specify a file containing an MRI-compatible script using the ‘`-c`’ command-line option.

Each command in an MRI-compatible script occupies its own line; each command line starts with the keyword that identifies the command (though blank lines are also allowed for punctuation). If a line of an MRI-compatible script begins with an unrecognized keyword, `ld` issues a warning message, but continues processing the script. Lines beginning with ‘`*`’ are comments.

You can write these commands using all upper-case letters, or all lower case; for example, ‘`chip`’ is the same as ‘`CHIP`’. The following list shows only the upper-case form of each command.

`ABSOLUTE` *`secname`*
`ABSOLUTE` *`secname`*`,` *`secname`*`, ...` *`secname`*

Normally, `ld` includes in the output file all sections from all the input files. However, in an MRI-compatible script, you can use the `ABSOLUTE` command to restrict the sections that will be present in your output program. If the `ABSOLUTE` command is used at all in a script, then only the sections named explicitly in `ABSOLUTE` commands will appear in the linker output. You can still use other input sections (whatever you select on the command line, or using `LOAD`) to resolve addresses in the output file.

`ALIAS` *`out-secname`*`,` *`in-secname`*

Use this command to place the data from input section *`in-secname`* in a section called *`out-secname`* in the linker output file. *`in-secname`* may be an integer.

`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`

This command does nothing whatever; it's only accepted for compatibility.

`FORMAT` *`output-format`*

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

- 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 command-line option '`-o`' or the general script language command `OUTPUT`.

`ORDER` *`secname, secname, ... secname`*

`ORDER` *`secname secname secname`*

Normally, `ld` orders the sections in its output file in the order in which they first appear in the input files. In an MRI-compatible script, you can override this ordering with the `ORDER` command. The sections you list with `ORDER` will appear first in your output file, in the order specified.

`PUBLIC` *`name=expression`*

`PUBLIC` *`name, expression`*

`PUBLIC` *`name expression`*

Supply a value (*`expression`*) for external symbol name used in the linker input files.

`SECT` *`secname, expression`*

`SECT` *`secname=expression`*

`SECT` *`secname expression`*

You can use any of these three forms of the `SECT` command to specify the start address (*`expression`*) for `section` *`sec-name`*. If you have more than one `SECT` statement for the same *`sec-name`*, only the *first* sets the start address.

GNUPro™ Toolkit

The GNU Binary Utilities

July, 1998

98r1

CYGNUS

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

1

Overview of the GNU binary utilities

The following documentation contains basic descriptions for the GNU binary utilities:

`ar`
: Creates, modifies, and extracts from archives; see “ar” on page 259

`nm`
: Lists symbols from object files; see “nm” on page 265

`objcopy`
: Copies and translates object files; see “objcopy” on page 269

`objdump`
: Displays information from object files; see “objdump” on page 274

`ranlib`
: Generates index to archive contents; see “ranlib” on page 279

`size`
: Lists file section sizes and total size; see “size” on page 280

`strings`
: Lists printable strings from files; see “strings” on page 282

`strip`
: Discards symbols; see “strip” on page 283

`c++filt`
: *Demangles* encoded C++ symbols; see “c++filt” on page 285

`addr2line`
: Converts addresses into file names and line numbers; see “addr2line” on page 287

`nlmconv`
: Converts object code into a Netware Loadable Module (NLM); see “nlmconv” on page 289

`windres`
: Manipulates Windows resources; see “windres” on page 291

ar

```
ar [-]p[mod [relpos]] archive [member ...]
ar -M [ <mri-script ]
```

The GNU `ar` program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called *members* of the archive).

The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

GNU `ar` can maintain archives whose members have names of any length; however, depending on how `ar` is configured on your system, a limit on member-name length may be imposed for compatibility with archive formats maintained with other tools. If it exists, the limit is often 15 characters (typical of formats related to `a.out`) or 16 characters (typical of formats related to coff).

`ar` is considered a binary utility because archives of this sort are most often used as libraries holding commonly needed subroutines.

`ar` creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier '`s`'. Once created, this index is updated in the archive whenever `ar` makes a change to its contents (save for the '`q`' update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use '`nm -s`' or '`nm --print-armap`' to list this index table. If an archive lacks the table, another form of `ar` called `ranlib` can be used to add only the table.

GNU `ar` is designed to be compatible with two different facilities. You can control its activity using command-line options like the different varieties of `ar` on Unix systems; or, if you specify the single command-line option '`-M`', you can control it with a script supplied via standard input, like the MRI *librarian* program.

Controlling `ar` on the command line

```
ar [-]p[mod [relpos]] archive [member ...]
```

When you use `ar` in the Unix style, `ar` insists on at least two arguments to execute: one keyletter specifying the *operation* (optionally accompanied by other keyletters specifying *modifiers*), and the archive name to act on.

Most operations can also accept further member arguments, specifying particular files to operate on.

GNU `ar` allows you to mix the operation code *p* and modifier flags *mod* in any order, within the first command-line argument.

If you wish, you may begin the first command-line argument with a dash.

The *p* keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

`d`

Delete modules from the archive. Specify the names of modules to be deleted as *`member`*`...`; the archive is untouched if you specify no files to delete.

If you specify the ‘`v`’ modifier, `ar` lists each module as it is deleted.

`m`

Move members in an archive.

The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

If no modifiers are used with `m`, any members you name in the *`member`* arguments are moved to the *end* of the archive; you can use the ‘`a`’, ‘`b`’, or ‘`i`’ modifiers to move them to a specified place instead.

`p`

Print the specified members of the archive, to the standard output file. If the ‘`v`’ modifier is specified, show the member name before copying its contents to standard output.

If you specify no *`member`* arguments, all the files in the archive are printed.

`q`

Quick append; add the files *`member`*`...` to the end of *`archive`*, without checking for replacement.

The modifiers ‘`a`’, ‘`b`’, and ‘`i`’ do *not* affect this operation; new members are always placed at the end of the archive. The modifier ‘`v`’ makes `ar` list each file as it is appended. Since the point of this operation is speed, the archive’s symbol table index is not updated, even if it already existed; you can use ‘`ar s`’ or `ranlib` explicitly to update the symbol table index.

`r`

Replacement; inserts the files *`member`*`...` into *`archive`*. This operation differs from ‘`q`’ in that any previously existing members are deleted if their names match those being added.

If one of the files named in *`member`*`...` does not exist, `ar` displays an error message, and leaves undisturbed any existing members of the archive matching that name.

By default, new members are added at the end of the file; but you may use one of the modifiers ‘`a`’, ‘`b`’, or ‘`i`’ to request placement relative to some existing member.

The modifier ‘`v`’ used with this operation elicits a line of output for each file inserted, along with one of the letters '`a`' or '`r`' to indicate whether the file was appended (no old member deleted) or replaced.

`t`

Display a *table* listing the contents of *`archive`*, or those of the files listed in *`member`*`...` that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the '`v`' modifier.

If you do not specify a *`member`*, all files in the archive are listed.

If there is more than one file with the same name (for instance, '`fie`') in an archive (for instance, '`b.a`'), '`ar t b.a fie`' lists only the first instance; to see them all, you must ask for a complete listing—in the earlier example, '`ar t b.a`'.

`x`

Extract members (named *`member`*) from the archive. You can use the '`v`' modifier with this operation, to request that `ar` list each name as it extracts it. If you do not specify a *`member`*, all files in the archive are extracted.

A number of modifiers (*`mod`*) may immediately follow the *`p`* keyletter, to specify variations on an operation's behavior:

`a`

Add new files *after* an existing member of the archive. If you use the modifier '`a`', the name of an existing archive member must be present as the `relpos` argument, before the archive specification.

`b`

Add new files *before* an existing member of the archive. If you use the modifier '`b`', the name of an existing archive member must be present as the `relpos` argument, before the archive specification. (same as '`i`').

`c`

Create the archive. The specified *`archive`* is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

`f`

Truncate names in the archive. GNU `ar` will normally permit file names of any length. This will cause it to create archives which are not compatible with the native `ar` program on some systems. If this is a concern, the '`f`' modifier may be used to truncate file names when putting them in the archive.

`i`

Insert new files *before* an existing member of the archive. If you use the modifier '`i`', the name of an existing archive member must be present as the *`relpos`* argument, before the *`archive`* specification. (same as '`b`').

`l`
: This modifier is accepted but not used.

`o`
: Preserve the *original* dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

`s`
: Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running '`ar s`' on an archive is equivalent to running '`ranlib`' on it.

`u`
: Normally, '`ar r`' ... inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The '`u`' modifier is allowed only for the operation '`r`' (*replace*). In particular, the combination '`qu`' is not allowed, since checking the timestamps would lose any speed advantage from the operation '`q`'.

`v`
: This modifier requests the verbose version of an operation. Many operations display additional information, such as file-names processed, when the modifier '`v`' is appended.

`V`
: This modifier shows the version number of `ar`.

Controlling `ar` with a script

```
ar -M [ < script ]
```

If you use the single command-line option '`-M`' with `ar`, you can control its operation with a rudimentary command language. This form of `ar` operates interactively if standard input is coming directly from a terminal. During interactive use, `ar` prompts for input (the prompt is '`AR >`'), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and `ar` abandons execution (with a nonzero exit code) on any error.

The `ar` command language is *not* designed to be equivalent to the command-line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to GNU `ar` for developers who already have scripts written for the MRI *librarian* program.

The syntax for the `ar` command language is straightforward:

- Commands are recognized in upper or lower case; for example, `LIST` is the same as `list`. In the following descriptions, commands are shown in upper case for clarity.
- A single command may appear on each line; it is the first word on the line.
- Empty lines are allowed, and have no effect.
- Comments are allowed; text after either of the characters '`*`' or '`;`' is ignored.
- Whenever you use a list of names as part of the argument to an `ar` command, you can separate the individual names with either commas or blanks. Commas are shown in the following explanations, for clarity.
- '`+`' is used as a line continuation character; if '`+`' appears at the end of a line, the text on the following line is considered part of the current command.

The following are the commands you can use in `ar` scripts, or when using `ar` interactively. Three of them have special significance:

`OPEN` or `CREATE` specify a *current* archive, which is a temporary file required for most of the other commands.

`SAVE` commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.

`ADDLIB` *`archive`*
`ADDLIB` *`archive`*`(`*`module`*`,` *`module`*`,` *`...module`*`)`

Add all the contents of archive(or, if specified, each named *`module`* from *`archive`*) to the current archive.

Requires prior use of `OPEN` or `CREATE`.

`ADDMOD` *`member`*`,` *`member`*`,` *`...member`*

Add each named `member` as a module in the current archive.

Requires prior use of `OPEN` or `CREATE`.

`CLEAR`

Discard the contents of the current archive, canceling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.

`CREATE` *`archive`*

Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as *`archive`* until you use `SAVE`. You can overwrite existing archives; similarly, the contents of any existing file named *`archive`* will not be destroyed until `SAVE`.

`DELETE` *`module`*`,` *`module`*`,` *`...module`*

Delete each listed *`module`* from the current archive.

Equivalent to '`ar -d` *`archive module ...module`*'.

Requires prior use of `OPEN` or `CREATE`.

`DIRECTORY` *`archive`*`(`*`module`*`, ...`*`module`*`)`
`DIRECTORY` *`archive`*`(` *`module`*`, ...`*`module`*`)` *`outputfile`*

List each named *`module`* present in *`archive`*. The separate command `VERBOSE` specifies the form of the output: when verbose output is off, output is like that of '`ar -t` *`archive module`* `...`'. When verbose output is on, the listing is like '`ar -tv` *`archive module`* `...`'.

Output normally goes to the standard output stream; however, if you specify *`outputfile`* as a final argument, `ar` directs the output to that file.

`END`

Exit from `ar`, with a `0` exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last `SAVE` command, those changes are lost.

`EXTRACT` *`module`*`,` *`module`*`, ...`*`module`*

Extract each named *`module`* from the current archive, writing them into the current directory as separate files.

Equivalent to '`ar -x` *`archive module`* `...`'.

Requires prior use of `OPEN` or `CREATE`.

`LIST`

Display full contents of the current archive, in *verbose* style regardless of the state of `VERBOSE`. The effect is like '`ar tv` *`archive`*'. This single command is a GNU `ld` enhancement, rather than present for MRI compatibility.

Requires prior use of `OPEN` or `CREATE`.

`OPEN archive`

Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect *`archive`* until you next use `SAVE`.

`REPLACE module, module, ...module`

In the current archive, replace each existing *`module`* (named in the `REPLACE` arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist. Requires prior use of `OPEN` or `CREATE`.

`VERBOSE`

Toggle an internal flag governing the output from `DIRECTORY`. When the flag is on, `DIRECTORY` output matches output from '`ar -tv` ' ...

`SAVE`

Commit your changes to the current archive, and actually save it as a file with the name specified in the last `CREATE` or `OPEN` command.

Requires prior use of `OPEN` or `CREATE`.

nm

```
nm [ -a | --debug-syms ] [ -g | --extern-only ]
   [ -B ] [ -C | --demangle ] [ -D | --dynamic ]
   [ -s | --print-armap ] [ -A | -o | --print-file-name ]
   [ -n | -v | --numeric-sort ] [ -p | --no-sort ]
   [ -r | --reverse-sort ] [ --size-sort ]
   [ -u | --undefined-only ]
   [-t radix | --radix= radix ]
   [ -P | --portability ] [ --target= bfdname ]
   [-f format | --format= format ]
   [ --defined-only ] [ -l | --line-numbers ]
   [ --no-demangle ] [ -V | --version ]
   [ --help ] [ objfile ...]
```

GNU `nm` lists the symbols from object files *`objfile`* ... If no object files are listed as arguments, `nm` assumes '`a.out`'.

For each symbol, `nm` shows:

- The symbol value, in the radix selected by the following options, or hexadecimal by default.
- The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

  `A`

  The symbol's value is absolute, and will not be changed by further linking.

  `B`

  The symbol is in the uninitialized data section (known as BSS).

  `C`

  The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references. For more details on common symbols, see the discussion of –`warn-common` in "Command line options for ld" on page 192 in *Using LD*.

  `D`

  The symbol is in the initialized data section.

  `G`

  The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global `int` variable as opposed to a large global `array`.

  `I`

  The symbol is an indirect reference to another symbol. This is a GNU extension to the `a.out` object file format which is rarely used.

`N`
: The symbol is a debugging symbol.

`R`
: The symbol is in a read only data section.

`S`
: The symbol is in an uninitialized data section for small objects.

`T`
: The symbol is in the text (code) section.

`U`
: The symbol is undefined.

`W`
: The symbol is weak. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

`-`
: The symbol is a `stabs` symbol in an `a.out` object file. In this case, the next values printed are the `stabs other` field, the `stabs desc` field, and the `stab` type. `stabs` symbols are used to hold debugging information; for more information, see ".stabd, .stabn, .stabs" on page 83 in *Using AS*.

`?`
: The symbol type is unknown, or object file format specific.

- The symbol name.

The long and short forms of options, shown here as alternatives, are equivalent.

`-A`
`-o`
`--print-file-name`
: Precede each symbol by the name of the input file (or archive element) in which it was found, rather than identifying the input file once only, before all of its symbols.

`-a`
`--debug-syms`
: Display all symbols, even debugger-only symbols; normally these are not listed.

`-B`
: The same as '`--format=bsd`' (for compatibility with the MIPS `nm`).

`-C`
`--demangle`
: Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++

function names readable. See “c++filt” on page 285 for more information on demangling.

`--no-demangle`
: Do not *demangle* low-level symbol names. This is the default.

`-D`
`--dynamic`
: Display the dynamic symbols rather than the normal symbols. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`-f` *`format`*
`--format=`*`format`*
: Use the output format, *`format`*, which can be `bsd`, `sysv`, or `posix`. The default is `bsd`. Only the first character of *`format`* is significant; it can be either upper or lower case.

`-g`
`--extern-only`
: Display only external symbols.

`-l`
`--line-numbers`
: For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry that refers to the symbol. If line number information can be found, print it after the other symbol information.

`-n`
`-v`
`--numeric-sort`
: Sort symbols numerically by their addresses, rather than alphabetically by their names.

`-p`
`--no-sort`
: Do not bother to sort the symbols in any order; print them in the order encountered.

`-P`
`--portability`
: Use the POSIX.2 standard output format instead of the de-fault format. Equivalent to ‘`-f posix`’.

`-s`
`--print-armap`
: When listing symbols from archive members, include the index: a mapping (stored in the archive by `ar` or `ranlib`) of which modules contain definitions for which names.

`-r`

`--reverse-sort`
: Reverse the order of the sort (whether numeric or alphabetic); let the last come first.

`--size-sort`
: Sort symbols by `size`. `size` is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.

`-t radix`

`--radix=`*`radix`*
: Use *`radix`* as the radix for printing the symbol values. It must be ‘`d`’ for decimal, ‘`o`’ for octal, or ‘`x`’ for hexadecimal.

`--target=`*`bfdname`*
: Specify an object code format other than your system’s default format. See “Target selection” on page 296, for more information.

`-u`

`--undefined-only`
: Display only undefined symbols (those external to each object file).

`--defined-only`
: Display only defined symbols for each object file.

`-V`

`--version`
: Show the version number of `nm` and exit.

`--help`
: Show a summary of the options to `nm` and exit.

objcopy

```
objcopy [ -F bfdname | --target=bfdname ]
        [ -I bfdname | --input-target=bfdname ]
        [ -O bfdname | --output-target=bfdname ]
        [ -S | --strip-all ] [ -g | --strip-debug ]
        [ -K symbolname | --keep-symbol=symbolname ]
        [ -N symbolname | --strip-symbol=symbolname ]
        [ -L symbolname | --localize-symbol=symbolname ]
        [-W symbolname | --weaken-symbol=symbolname ]
        [ -x | --discard-all ] [ -X | --discard-locals ]
        [ -b byte | --byte=byte ]
        [ -i interleave | --interleave=interleave ]
        [ -R sectionname | --remove-section=sectionname ]
        [ -p | --preserve-dates ] [ --debugging ]
        [ --gap-fill=val ] [ --pad-to=address ]
        [ --set-start=val ] [ --adjust-start=incr ]
        [ --adjust-vma=incr ]
        [ --adjust-section-vma=section{=,+,-}val ]
        [ --adjust-warnings ] [ --no-adjust-warnings ]
        [ --set-section-flags=section=flags ]
        [ --add-section=sectionname=filename ]
        [ --change-leading char ] [--remove-leading-char ]
        [ --weaken ]
        [ -v | --verbose ] [ -V | --version ] [ --help ]
        infile [outfile]
```

The GNU `objcopy` utility copies the contents of an object file to another. `objcopy` uses the GNU BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of `objcopy` is controlled by command-line options.

`objcopy` creates temporary files to do its translations and deletes them afterward. `objcopy` uses BFD to do all its translation work; it has access to all the formats described in BFD and is able to recognize most formats without being told explicitly. See “BFD” on page 245 and “The BFD canonical object-file format” on page 249 in *Using LD*.

`objcopy` can be used to generate S-records by using an output target of ‘`srec`’ (use ‘`-O srec`’).

`objcopy` can be used to generate a raw binary file by using an output target of ‘`binary`’ (meaning ‘`-O binary`’). When `objcopy` generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the virtual address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use `-S` to remove sections containing debugging information. In some cases '`-R`' will be useful to remove sections containing information that is not needed by the binary file.

`infile`
`outfile`
The source and output files, respectively. If you do not specify *`outfile`*, `objcopy` creates a temporary file and destructively renames the result with the name of *`infile`*.

`-I` *`bfdname`*
`--input-target=`*`bfdname`*
Consider the source file's object format to be *`bfdname`*, rather than attempting to deduce it. See "Target selection" on page 296 for more information.

`-O` *`bfdname`*
`--output-target=`*`bfdname`*
Write the output file using the object format, *`bfdname`*. See "Target selection" on page 296 for more information.

`-F` *`bfdname`*
`--target=`*`bfdname`*
Use *`bfdname`* as the object format for both the input and the output file; i.e., simply transfer data from source to destination with no translation. See "Target selection" on page 296 for more information.

`-R` *`sectionname`*
`--remove-section=`*`sectionname`*
Remove any section named *`sectionname`* from the output file. This option may be given more than once.

NOTE: Using this option inappropriately may make the output file unusable.

`-S`
`--strip-all`
Do not copy relocation and symbol information from the source file.

`-g`
`--strip-debug`
Do not copy debugging symbols from the source file.

`--strip-unneeded`
Strip all symbols that are not needed for relocation processing.

`-K` *`symbolname`*
`--keep-symbol=`*`symbolname`*
Copy only symbol *`symbolname`* from the source file. This option may be given more than once.

`-N` *`symbolname`*
`--strip-symbol=`*`symbolname`*
Do not copy symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with strip options other than `-K`.

`-L symbolname`
`--localize-symbol=`*`symbolname`*
Make `symbol` *`symbolname`* local to the file, so that it is not visible externally. This option may be given more than once.

`-W symbolname`
`--weaken-symbol=`*`symbolname`*
Make `symbol` *`symbolname`* weak. This option may be given more than once.

`-x`
`--discard-all`
Do not copy non-global symbols from the source file.

`-X`
`--discard-locals`
Do not copy compiler-generated local symbols. (These usually start with ‘`L`’ or ‘`.`’.)

`-b` *`byte`*
`--byte=`*`byte`*
Keep only every *`byte`* of the input file (header data is not affected). *`byte`* can be in the range from 0 to *`interleave`*-1, where interleave is given by the ‘`-i`’ or ‘`--interleave`’ option, or the default of 4. This option is useful for creating files to program ROM . It is typically used with an `srec` output target.

`-i` *`interleave`*
`--interleave=`*`interleave`*
Only copy one out of every *`interleave`* bytes. Select which byte to copy with the ‘`-b`’ or ‘`--byte`’ option. The default is 4. `objcopy` ignores this option if you do not specify either ‘`-b`’ or ‘`--byte`’.

`-p`
`--preserve-dates`
Set the access and modification dates of the output file to be the same as those of the input file.

`--debugging`
Convert debugging information, if possible. This is not the default because only certain debugging formats are supported, and the conversion process can be time consuming.

`--gap-fill` *`val`*
Fill gaps between sections with *`val`*. This is done by increasing the size of the section with the lower address, and filling in the extra space created with *`val`*.

`--pad-to` *`address`*
: Pad the output file up to the virtual address *`address`*. This is done by increasing the size of the last section. The extra space is filled in with the value specified by '`--gap-fill`' (default zero).

`--set-start` *`val`*
: Set the address of the new file to *`val`*. Not all object file formats support setting the start address.

`--adjust-start` *`incr`*
: Adjust the start address by adding *`incr`*. Not all object file formats support setting the start address.

`--adjust-vma` *`incr`*
: Adjust the address of all sections, as well as the start address, by adding *`incr`*. Some object file formats do not permit section addresses to be changed arbitrarily. Note that this does not relocate the sections; if the program expects sections to be loaded at a certain address, and this option is used to change the sections such that they are loaded at a different address, the program may fail.

`--adjust-section-vma` *`section`*`{=,+,-}`*`val`*
: Set or adjust the address of the named *`section`*. If '`=`' is used, the section address is set to *`val`*. Otherwise, *`val`* is added to or subtracted from the section address. See the comments under '`--adjust-vma`'. If *`section`* does not exist in the input file, a warning will be issued, unless '`--no-adjust-warnings`' is used.

`--adjust-warnings`
: If '`--adjust-section-vma`' is used, and the named section does not exist, issue a warning. This is the default.

`--no-adjust-warnings`
: Do not issue a warning if '`--adjust-section-vma`' is used, even if the named section does not exist.

`--set-section-flags` *`section`*`=`*`flags`*
: Set the flags for the named section. The *`flags`* argument is a comma separated string of flag names. The recognized names are '`alloc`', '`load`', '`readonly`', '`code`', '`data`', and '`rom`'. Not all flags are meaningful for all object file formats.

`--add-section` *`sectionname`*`=`*`filename`*
: Add a new section named *`sectionname`* while copying the file. The contents of the new section are taken from the file *`filename`*. The size of the section will be the size of the file. This option only works on file formats which can support sections with arbitrary names.

`--change-leading-char`
: Some object file formats use special characters at the start of symbols. The most common such character is underscore, which compilers often add before every symbol. This option tells `objcopy` to change the leading character of every symbol when it converts between object file formats. If the object file formats use the same leading character, this option has no effect. Otherwise, it will add a character, or remove a character, or change a character, as appropriate.

`--remove-leading-char`
: If the first character of a global symbol is a special symbol leading character used by the object file format, remove the character. The most common symbol leading character is underscore. This option will remove a leading underscore from all global symbols. This can be useful if you want to link together objects of different file formats with different conventions for symbol names.

`--weaken`
: Change all global symbols in the file to be weak. This can be useful when building an object that will be linked against other objects using the `-R` option to the linker. This option is only effective when using an object file format that supports weak symbols.

`-V`
`--version`
: Show the version number of `objcopy`.

`-v`
`--verbose`
: Verbose output: list all object files modified. In the case of archives, '`objcopy -V`' lists all members of the archive.

`--help`
: Show a summary of the options to `objcopy`.

objdump

```
objdump [ -a | --archive-headers ]
        [ -b bfdname | --target=bfdname ] [ --debugging ]
        [ -d | --disassemble ] [ -D | --disassemble-all ]
        [ -EB | -EL | --endian={big | little } ]
        [ -f | --file-headers ]
        [ -h | --section-headers | --headers ]
        [ -i | --info ]
        [ -j section | --section=section ]
        [ -l | --line-numbers ] [ -S | --source ]
        [ -m machine | --architecture=machine ]
        [ -r | --reloc ] [ -R | --dynamic-reloc ]
        [ -s | --full-contents ] [ --stabs ]
        [ -t | --syms ] [ -T | --dynamic-syms ]
        [ -x | --all-headers ]
        [ -w | --wide ] [ --start-address=address ]
        [ --stop-address=address ]
        [ --prefix-addresses] [ --show-raw-insn ]
        [ --adjust-vma=offset ]
        [ --version ]
        [ --help ]
        objfile...
```

`objdump` displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

`objfile``...` are the object files to be examined. When you specify archives, `objdump` shows information on each of the member object files.

The long and short forms of options, shown here as alternatives, are equivalent. At least one option besides '`-l`' must be given.

`-a`
`--archive-header`

If any of the objfile files are archives, display the archive header information (in a format similar to '`ls -l`'). Besides the information you could list with '`ar tv`', '`objdump -a`' shows the object file format of each archive member.

`--adjust-vma=`*`offset`*

When dumping information, first add *`offset`* to all the section addresses. This is useful if the section addresses do not correspond to the symbol table, which can happen when putting sections at particular addresses when using a format that can not represent section addresses, such as `a.out`.

`-b bfdname`
`--target=bfdname`

Specify that the object-code format for the object files is BFD-name. This option may not be necessary; `objdump` can automatically recognize many formats.

```
objdump -b oasys -m vax -h fu.o
```

The previous example displays summary information from the section headers (‘`-h`’) of ‘`fu.o`’, which is explicitly identified (‘`-m`’) as a Vax object file in the format produced by Oasys compilers. You can list the formats available with the ‘`-i`’ option. See “Target selection” on page 296 for more information.

`-C`
`--demangle`

Decode (demangle) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See “c++filt” on page 285 for more information on demangling.

`--debugging`

Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C like syntax. Only certain types of debugging information have been implemented.

`-d`
`--disassemble`

Display the assembler mnemonics for the machine instructions from *objfile*. This option only disassembles those sections which are expected to contain instructions.

`-D`
`--disassemble-all`

Like ‘`-d`’, but disassembles the contents of all sections, not just those expected to contain instructions.

`--prefix-addresses`

When disassembling, print the complete address on each line. This is the older disassembly format.

`--disassemble-zeroes`

Normally the disassembly output will skip blocks of zeroes.

This option directs the disassembler to disassemble those blocks, just like any other data.

`-EB`
`-EL`
`--endian={big | little}`

Specify the endianness of the object files. This only affects disassembly. This can be useful when disassembling a file format that does not describe endianness information, such as S-records.

`-f`
`--file-header`
Display summary information from the overall header of each of the *objfile* files.

`-h`
`--section-header`
`--header`
Display summary information from the section headers of the object file.

File segments may be relocated to nonstandard addresses, for example by using the '`-Ttext`', '`-Tdata`', or '`-Tbss`' options to `ld`. However, some object file formats, such as `a.out`, do not store the starting address of the file segments. In those situations, although `ld` relocates the sections correctly, using '`objdump -h`' to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.

`--help`
Print a summary of the options to `objdump` and exit.

`-i`
`--info`
Display a list showing all architectures and object formats available for specification with '`-b`' or '`-m`'.

`-j` *name*
`--section=`*name*
Display information only for section *name*.

`-l`
`--line-numbers`
Label the display (using debugging information) with the filename and source line numbers corresponding to the object code shown. Only useful with '`-d`' or '`-D`'.

`-m` *machine*
`--architecture=`*machine*
Specify that the object files, *objfile*, are for the intended architecture, *machine*. You can list available architectures using the '`-i`' option.

`-r`
`--reloc`
Print the relocation entries of the file. If used with '`-d`' or '`-D`'options, the relocations are printed interspersed with the disassembly.

`-R`
`--dynamic-reloc`
Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`-s`
`--full-contents`
: Display the full contents of any sections requested.

`-S`
`--source`
: Display source code intermixed with disassembly, if possible.

  Implies ‘`-d`’.

`--show-raw-insn`
: When disassembling instructions, print the instruction in hex as well as in symbolic form. Not all targets handle this correctly yet.

`--no-show-raw-insn`
: When disassembling instructions, do not print the instruction bytes. This is the default when using `--prefix-addresses`.

`--stabs`
: Display the full contents of any sections requested. Display the contents of the `.stab` and `.stab.index` and `.stab.excl` sections from an ELF file. This is only useful on systems (such as Solaris 2.0) in which `.stab` debugging symbol-table entries are carried in an ELF section. In most other file formats, debugging symbol-table entries are interleaved with linkage symbols, and are visible in the ‘`--syms`’ output. For more information on `stabs` symbols, see “.stabd, .stabn, .stabs” on page 83 in *Using AS*.

`--start-address=`*`address`*
: Start displaying data at the specified address. This affects the output of the `-d`, `-r` and `-s` options.

`--stop-address=`*`address`*
: Stop displaying data at the specified address. This affects the output of the `-d`, `-r` and `-s` options.

`-t`
`--syms`
: Print the symbol table entries of the file. This is similar to the information provided by the ‘`nm`’ program.

`-T`
`--dynamic-syms`
: Print the dynamic symbol table entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries. This is similar to the information provided by the ‘`nm`’ program when given the ‘`-D`’ (‘`--dynamic`’) option.

`--version`
: Print the version number of `objdump` and exit.

`-x`
`--all-header`
Display all available header information, including the symbol table and relocation entries.

Using ‘`-x`’ is equivalent to specifying all of ‘`-a -f -h -r -t`’.

`-w`
`--wide`
Format some lines for output devices having more than 80 columns.

ranlib

```
ranlib [-vV] archive
```

`ranlib` generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file.

You may use '`nm -s`' or '`nm --print-armap`' to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

The GNU `ranlib` program is another form of GNU `ar`; running `ranlib` is completely equivalent to executing '`ar -s`'. See "ar" on page 259.

`-v`
`-V`
: Show the version number of `ranlib`.

size

```
size  [ -A | -B | --format=compatibility ]
      [ --help ] [ -d | -o | -x | --radix=number ]
      [ --target=bfdname ] [ -V | --version ]
      objfile...
```

The GNU `size` utility lists the section sizes—and the total size—for each of the object or archive files *`objfile`* in its argument list. By default, one line of output is generated for each object file or each module in an archive.

`objfile` `...` are the object files to be examined.

The command line options have the following meanings.

`-A`
`-B`
`--format=`*`compatibility`*

Using one of these options, you can choose whether the output from GNU `size` resembles output from System V `size` (using '`-A`', or '`--format=sysv`'), or Berkeley `size` (using '`-B`', or '`--format=berkeley`'). The default is the one-line format similar to Berkeley's.

The following is an example of the Berkeley (default) format of output from `size`.

```
size --format=Berkeley ranlib size
text    data     bss    dec     hex     filename
294880  81920    11592  388392  5ed28   ranlib
294880  81920    11888  388688  5ee50   size
```

The following example shows the same data, but displayed closer to System V conventions.

```
size --format=SysV ranlib size
ranlib :
section      size      addr
.text        294880    8192
.data        81920     303104
.bss         11592     385024
Total        388392

size :
section      size      addr
.text        294880    8192
.data        81920     303104
.bss         11888     385024
Total        388688
```

`--help`

Shows a summary of acceptable arguments and options.

`-d`
`-o`
`-x`
`--radix=`*`number`*

Using one of these options, you can control whether the size of each section is given: in decimal (‘`-d`’, or ‘`--radix=10`’); in octal (‘`-o`’, or ‘`--radix=8`’); or, in hexadecimal (‘`-x`’, or ‘`--radix=16`’).

In ‘`--radix=`*`number`*’, only the three values (8, 10, 16) are supported. The total size is always given in two radices; decimal and hexadecimal for ‘`-d`’ or ‘`-x`’ output; or octal and hexadecimal if you’re using ‘`-o`’.

`--target=`*`bfdname`*

Specify that the object-code format for *`objfile`* is *`bfdname`*. This option may not be necessary; `size` can automatically recognize many formats. See “Target selection” on page 296 for more information.

`-V`
`--version`

Display the version number of `size`.

strings

```
strings  [-afov] [-min-len] [-n min-len] [-t radix] [-]
         [--all] [--print-file-name] [--bytes=min-len]
         [--radix=radix] [--target=bfdname]
         [--help] [--version] file ...
```

For each *file* given, GNU `strings` prints the printable character sequences that are at least 4 characters long (or the number given with the following options) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

`strings` is mainly useful for determining the contents of non-text files.

`-a`
`--all`
`-`
: Do not scan only the initialized and loaded sections of object files; scan the whole files.

`-f`
`--print-file-name`
: Print the name of the file before each string.

`--help`
: Print a summary of the program usage on the standard output and exit.

`-`*`min-len`*
`-n` *`min-len`*
`--bytes=`*`min-len`*
: Print sequences of characters that are at least *min-len* characters long, instead of the default 4.

`-o`
: Like '`-t o`'. Some other versions of strings have '`-o`' act like '`-t d`'. Since we can not be compatible with both ways, we simply chose one.

`-t radix`
`--radix=`*`radix`*
: Print the offset within the file before each string. The single character argument specifies the radix of the offset—'`o`' for octal, '`x`' for hexadecimal, or '`d`' for decimal.

`--target=`*`bfdname`*
: Specify an object code format other than your system's default format. See "Target selection" on page 296 for more information.

`-v`
`--version`
: Print the program version number on the standard output and exit.

strip

```
strip  [ -F bfdname | --target=bfdname ]
        [ -I bfdname | --input-target=bfdname ]
        [ -O bfdname | --output-target=bfdname ]
        [ -s | --strip-all ] [ -S | -g | --strip-debug ]
        [ -K symbolname | --keep-symbol=symbolname ]
        [ -N symbolname | --strip-symbol=symbolname ]
        [ -x | --discard-all ] [ -X | --discard-locals ]
        [ -R sectionname | --remove-section=sectionname ]
        [-o file ] [ -p | --preserve-dates ]
        [ -v | --verbose ] [ -V | --version ] [ --help ]
        objfile ...
```

GNU `strip` discards all symbols from object files *`objfile`*. The list of object files may include archives. At least one object file must be given. `strip` modifies the files named in its argument, rather than writing modified copies under different names.

`-F` *`bfdname`*
`--target=`*`bfdname`*
: Treat the original *`objfile`* as a file with the object code format *`bfdname`*, and rewrite it in the same format. See “Target selection” on page 296 for more information.

`--help`
: Show a summary of the options to `strip` and exit.

`-I` *`bfdname`*
`--input-target=`*`bfdname`*
: Treat the original *`objfile`* as a file with the object code format *`bfdname`*. See “Target selection” on page 296 for more information.

`-O` *`bfdname`*
`--output-target=`*`bfdname`*
: Replace *`objfile`* with a file in the output format, *`bfdname`*. See “Target selection” on page 296 for more information.

`-R` *`sectionname`*
`--remove-section=`*`sectionname`*
: Remove any section named *`sectionname`* from the output file. This option may be given more than once.

NOTE: Using this option inappropriately may make the output file unusable.

`-s`
`--strip-all`
: Remove all symbols.

`-g`
`-S`
`--strip-debug`
Remove debugging symbols only.

`--strip-unneeded`
Remove all symbols that are not needed for relocation processing.

`-K` *`symbolname`*
`--keep-symbol=`*`symbolname`*
Keep only symbol *`symbolname`* from the source file. This option may be given more than once.

`-N` *`symbolname`*
`--strip-symbol=`*`symbolname`*
Remove symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with `strip` options other than `-K`.

`-o` *`file`*
Put the stripped output in *`file`*, rather than replacing the existing file. When this argument is used, only one *`objfile`* argument may be specified.

`-p`
`--preserve-dates`
Preserve the access and modification dates of the file.

`-x`
`--discard-all`
Remove non-global symbols.

`-X`
`--discard-locals`
Remove compiler-generated local symbols. (These usually start with '`L`' or '`.`'.)

`-V`
`--version`
Show the version number for `strip`.

`-v`
`--verbose`
Verbose output: list all object files modified. In the case of archives, '`strip -v`' lists all members of the archive.

c++filt

```
c++filt [ -_ | --strip-underscores ]
         [ -n | --no-strip-underscores ]
         [ -sformat | --format=format ]
         [ --help ] [ --version ] [ symbol...]
```

The C++ language provides function overloading, which means that you can write many functions with the same name (providing each takes parameters of different types). All C++ function names are encoded into a low-level assembly label (this process is known as *mangling*). The `c++filt` program does the inverse mapping: it decodes (*demangles*) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

You can use `c++filt` to decipher individual symbols, using a declaration like the following example.

`c++filt` *`symbol`*

If no *`symbol`* arguments are given, `c++filt` reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

`-_`
`--strip-underscores`
: On some systems, both the C and C++ compilers put an underscore in front of every name. This option removes the initial underscore. Whether `c++filt` removes the underscore by default is target dependent.

`-n`
`--no-strip-underscores`
: Do not remove the initial underscore.

`-s` *`format`*
`--format=`*`format`*
: GNU `nm` can decode three different methods of mangling, used by different C++ compilers. The argument to this option selects which method it uses:

  `gnu`
  : The one used by the GNU compiler (the default method).

  `lucid`
  : The one used by the Lucid compiler.

  `arm`
  : The one specified by the ***C++ Annotated Reference Manual***.

`--help`
: Print a summary of the options to `c++filt` and exit.

`--version`
: Print the version number of `c++filt` and exit.

WARNING: `c++filt` is a developing utility, meaning that the details of its user interface are subject to change as C++ changes. In particular, a command-line option may be required in the future to decode a name passed as an argument on the command line; for example, `c++filt` *`symbol`* may in a future release become `c++filt` *`option symbol`*.

addr2line

```
addr2line [ -b bfdname | --target=bfdname ]
          [ -C | --demangle ]
          [ -e filename | --exe=filename ]
          [ -f | --functions ] [ -s | --basename ]
          [ -H | --help ] [ -V | --version ]
          [ addr addr ... ]
```

`addr2line` translates program addresses into file names and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which file name and line number are associated with a given address.

The executable to use is specified with the `-e` option. The default is `a.out`.

`addr2line` has two modes of operation.

In the first, hexadecimal addresses are specified on the command line, and `addr2line` displays the file name and line number for each address.

In the second, `addr2line` reads hexadecimal addresses from standard input, and prints the file name and line number for each address on standard output. In this mode, `addr2line` may be used in a pipe to convert dynamically chosen addresses.

The format of the output is `FILENAME:LINENO`. The file name and line number for each address is printed on a separate line. If the `-f` option is used, then each `FILENAME:LINENO` line is preceded by a `FUNCTIONNAME` line which is the name of the function containing the address.

If the file name or function name can not be determined, `addr2line` will print two question marks in their place. If the line number can not be determined, `addr2line` will print `0`.

The long and short forms of options, shown here as alternatives, are equivalent.

`-b bfdname`
`--target=`*`bfdname`*
: Specify that the object-code format for the object files is *`bfdname`*.

`-C`
`--demangle`
: Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See “c++filt” on page 285 for more information on demangling.

`-e filename`
`--exe=`*`filename`*
: Specify the name of the executable for which addresses should be translated. The default file is `a.out`.

`-f`
`--functions`

Display function names as well as file and line number information.

`-s`
`--basenames`

Display only the base of each file name.

nlmconv

nlmconv converts a relocatable object file into a NetWare Loadable Module.

warning:

nlmconv is not always built as part of the binary utilities, since it is only useful for NLM targets.

```
nlmconv [ -I bfdname | --input-target=bfdname ]
        [ -O bfdname | --output-target=bfdname ]
        [ -T headerfile | --header-file=headerfile ]
        [ -d | --debug] [ -l linker | --linker=linker ]
        [ -h | --help ] [ -V | --version ]
        infile outfile
```

nlmconv converts the relocatable 'i386' object file *infile* into the NetWare Loadable Module, *outfile*, optionally reading *headerfile* for NLM header information.

For instructions on writing the NLM command file language used in header files, see "linkers" or "NLMLINK" in particular, in the *NLM Development and Tools Overview*, which is part of the ***NLM Software Developer's Kit*** ("***NLM SDK***"), available from Novell, Inc.

nlmconv uses the GNU Binary File Descriptor library (BFD) to read *infile*; see the documentation for "BFD" on page 245 in *Using LD* for more information.

nlmconv can perform a link step. In other words, you can list more than one object file for input if you list them in the definitions file (rather than simply specifying one input file on the command line). In this case, nlmconv calls the linker for you.

-I *bfdname*

--input-target=*bfdname*

Object format of the input file. nlmconv can usually determine the format of a given file (so no default is necessary). See "Target selection" on page 296 for more information.

-O *bfdname*

--output-target=*bfdname*

Object format of the output file. nlmconv infers the output format based on the input format, e.g., for a 'i386' input file, the output format is 'nlm32-i386'. See "Target selection" on page 296 for more information.

-T *headerfile*

--header-file=*headerfile*

Reads *headerfile* for NLM header information. For instructions on writing the NLM command file language used in header files, see "linkers" or "NLMLINK" in particular, of the *NLM Development and Tools Overview*, which is part of the ***NLM Software Developer's Kit*** ("***NLM SDK***"), available from Novell, Inc.

`-d`
`--debug`
Displays (on standard error) the linker command line used by `nlmconv`.

`-l` *`linker`*
`--linker=`*`linker`*
Use *`linker`* for any linking. *`linker`* can be an absolute or a relative pathname.

`-h`
`--help`
Prints a usage summary.

`-V`
`--version`
Prints the version number for `nlmconv`.

windres

windres may be used to manipulate Windows resources.

WARNING: windres is not always built as part of the binary utilities, since it is only useful for Windows targets.

```
windres [options] [input-file] [output-file]
```

windres reads resources from an input file and copies them into an output file. Either file may be in one of three formats:

rc
: A text format read by the Resource Compiler.

res
: A binary format generated by the Resource Compiler.

coff
: A COFF object or executable.

The exact description of these different formats is available in documentation from Microsoft.

When windres converts from the rc format to the res format, it is acting like the Windows Resource Compiler. When windres converts from the res format to the coff format, it is acting like the Windows CVTRES program.

When windres generates an rc file, the output is similar but not identical to the format expected for the input. When an input rc file refers to an external filename, an output rc file will instead include the file contents.

If the input or output format is not specified, windres will guess based on the file name, or, for the input file, the file contents. A file with an extension of '.rc' will be treated as an rc file, a file with an extension of '.res' will be treated as a res file, and a file with an extension of '.o' or '.exe' will be treated as a coff format file.

If no output file is specified, windres will print the resources in rc format to standard output.

The normal use is for you to write an rc file, use windres to convert it to a COFF file, and then link the COFF file into your application.

This will make the resources described in the rc file available to Windows.

-i *filename*
--input *filename*
: The name of the input file. If this option is not used, then windres will use the first non-option argument as the input file name. If there are no non-option arguments, then windres will read from standard input. windres can not read a COFF file from standard input.

`-o` *`filename`*
`--output` *`filename`*
The name of the output file. If this option is not used, then `windres` will use the first non-option argument, after any used for the input file name, as the output file name. If there is no non-option argument, then `windres` will write to standard output. `windres` can not write a COFF file to standard output.

`-I` *`format`*
`--input-format` *`format`*
The input format to read. *`format`* may be `res`, `rc`, or `coff`. If no input format is specified, `windres` will guess.

`-O` *`format`*
`--output-format` *`format`*
The output format to generate. *`format`* may be `res`, `rc`, or `coff`. If no output format is specified, `windres` will guess.

`-F` *`target`*
`--target` *`target`*
Specify the BFD format to use for a COFF file as input or output. This is a BFD target name; you can use the `--help` option to see a list of supported targets. Normally `windres` will use the default format, which is the first one listed by the `--help` option.

`--preprocessor` *`program`*
When `windres` reads an `rc` file, it runs it through the C preprocessor first. This option may be used to specify the preprocessor to use, including any leading arguments. The default preprocessor argument uses the following commands and arguments:

```
gcc -E -xc-header DRC_INVOKED.
```

`--include-dir` *`directory`*
Specify an `include` directory to use when reading an `rc` file.

`windres` will pass this to the preprocessor as an `-I` option.

`windres` will also search this directory when looking for files named in the `rc` file.

`--define sym[=`*`val`*`]`
Specify a `-D` option to pass to the preprocessor when reading an `rc` file.

`--language` *`val`*
Specify the default language to use when reading an `rc` file.

`val` should be a hexadecimal language code. The low eight bits are the language, and the high eight bits are the sub-language.

`--help`
Prints a usage summary.

`--version`
: Prints the version number for `windres`.

`--yydebug`
: If `windres` is compiled with `YYDEBUG` defined as `1`, this will turn on parser debugging.

2

Selecting the target system

You can specify three aspects of the target system to the GNU binary file utilities, selecting each in several ways:

- as **target**; see “Target selection” on page 296 for discussions of the lists of ways to specify values in order of decreasing precedence for each specification
- as **architecture**; see “Architecture selection” on page 298 for the ways to manage the binary utilities on different processors
- as **linker emulation**,, a *personality* of the linker, giving the linker default values applying only to the linker; see “Linker emulation selection” on page 299

See “Overview of the GNU binary utilities” on page 257 for more documentation on the individual utilities.

Target selection

A *target* is an object file format. A given target may be supported for multiple architectures (see “Architecture selection” on page 298). A target selection may also have variations for different operating systems or architectures.

The commands to list valid values only list the values for which the programs you are running were configured. If they were configured with `--enable-targets=`*`all`*, the commands list most of the available values, but a few are left out; not all targets can be configured in at once because some of them can only be configured *native* (on hosts with the same type as the target system).

The command to list valid target values is ‘`objdump -i`’ (the first column of output contains the relevant information).

Some sample values are: ‘`a.out-hp300bsd`’, ‘`ecoff-littlemips`’, ‘`a.out-sunos-big`’.

You can also specify a target using a *configuration triplet*. This is the same sort of name that is passed to `configure` to specify a target.

When you use a configuration triplet as an argument, it must be fully *canonicalized*. You can see the canonical version of a triplet by running the shell script, `config.sub`, which is included with the sources.

Some sample configuration triplets are `m68k-hp-bsd`, `mips-dec-ultrix`, and `sparc-sun-sunos`.

`objdump` target

Ways to specify:

- command line option: ‘`-b`’ or‘`--target`’
- environment variable `GNUTARGET`
- deduced from the input file

`objcopy` and `strip` input target

Ways to specify:

- command line options, `-I`, `--input-target`, `-F`, or `--target`
- environment variable, `GNUTARGET`
- deduced from the input file

objcopy and strip Output target

Ways to specify:

- command line options, -O, --output-target, -F, or --target
- the input target (see "objcopy and strip input target" on page 296)
- environment variable, GNUTARGET
- deduced from the input file

nm, size, and strings target

Ways to specify:

- command line option, --target
- environment variable GNUTARGET
- deduced from the input file

Linker input target

Ways to specify:

- command line option, -b or --format (see "Command line options for ld" on page 192 in *Using LD*)
- script command, TARGET (see "Commands dealing with object file formats" on page 210 in *Using LD*)
- environment variable, GNUTARGET (see "Environment variables for ld" on page 204 in *Using LD*)
- the default target of the selected linker emulation (see "Linker emulation selection" on page 299).

Linker output target

Ways to specify:

- command line option, -oformat (see "Command line options for ld" on page 192 in *Using LD*)
- script command, OUTPUT_FORMAT (see "Commands dealing with object file formats" on page 210 in *Using LD*)
- the linker input target (see "Linker input target")

Architecture selection

An *architecture* is a type of CPU on which an object file is to run. Its name may contain a colon, separating the name of the processor family from the name of the particular processor.

The command to list valid architecture values is '`objdump -i`' (the second column contains the relevant information).

Sample values: `m68k:68020`, `mips:3000`, `sparc`.

`objdump` architecture

Ways to specify: `objdump`

- command line option: '`-m`' or'`--architecture`'
- deduced from the input file

`objcopy`, `nm`, `size`, `strings` architecture

Its specification is deduced from the input file.

Linker input architecture

Its specification is deduced from the input file.

Linker output architecture

Ways to specify:

- script command, `OUTPUT_ARCH` (see "Other linker script commands" on page 211 in *Using LD*)
- the default architecture from the linker output target (see "Target selection" on page 296)

Linker emulation selection

A *linker emulation* is a personality of the linker, giving the linker default values for the other aspects of the target system.

In particular, it consists of the linker script, the target and several *hook* functions (which are run at certain stages of the linking process to do special things that some targets require).

The command to list valid linker emulation values is `ld -V`.

Sample values: '`hp300bsd`', '`mipslit`', '`sun4`'.

Ways to specify:

- command line option, `-m` (see "Command line options for ld" on page 192 in *Using LD*)
- environment variable, `LDEMULATION`
- compiled-in `DEFAULT_EMULATION` from `Makefile` which comes from `EMUL` in '`config/`*`target`*`.mt`'

GNUPRO™ TOOLKIT

GNU Make

July, 1998
98r1

CYGNUS

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

1

Overview of `make`

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 options for make" on page 419
- "Quick reference to make" on page 479

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 305, all of which is overview. If you are familiar with other `make` programs, see "Features of GNU make" on page 457, "Special built-in target names" on page 340 and "Incompatibilities and missing features" on page 461 (which explains the few things GNU `make` lacks that other programs provide).

- "Introduction to makefiles" on page 305
- "Writing makefiles" on page 315
- "Writing rules" on page 325
- "Writing the commands in rules" on page 351
- "How to use variables" on page 369
- "Conditional parts of makefiles" on page 387
- "Functions for transforming text" on page 395

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 following documentation discusses the fundamentals of make.

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.

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 485.

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.

What a rule looks like

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

```
target ... : dependencies ...
        command
        ...
        ...
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘`clean`’ (see “Phony targets” on page 336).

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.

NOTE: You need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with dependencies and serves to create a target file if any of the dependencies change. However, the rule that specifies commands for the target need not have dependencies. For example, the rule containing the delete command associated with the target, `clean`, does not have dependencies.

A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the commands on the dependencies to create or update the target. A rule can also explain how and when to carry out an action. See “Writing rules” on page 325.

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`.

```
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
```

NOTE: We split each long line into two lines using backslash-newline (\); this is like using one long line, but is easier to read.

To use the previous sample makefile to create the executable file called ‘`edit`’, type: `make`.

To use this makefile to delete the executable file and all the object files from the directory, type: `make clean`.

See the sample makefile with “A simple makefile” on page 308, the targets include the executable file, `edit`, and the object files, `main.o` and `kbd.o`. The dependencies are files such as `main.c` and `defs.h`. In fact, each `.o` file is both a target and a dependency. Commands include `cc -c main.c` and `cc -c kbd.c`.

When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. In addition, any dependencies that are themselves automatically generated should 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. (Bear in mind that **`make`** does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All **`make`** does is execute the commands in the rule you have specified when the target file needs to be updated.)

The target `clean` is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, `clean` is not a dependency of any other rule. Consequently, **`make`** never does anything with it unless you tell it specifically. *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 336 for information about this kind of target. See “Errors in commands” on page 357 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 411.)

See the example shown with "A simple makefile" on page 308; 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 308; we had to list all the object files twice in the rule for ‘**`edit`**’ and we’ve repeated 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 369).

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 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 425.

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 311).

```
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 313is how we would write the makefile in actual practice. (The complications associated with 'clean' are described in "Phony targets" on page 336 and in "Errors in commands" on page 357.)

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 is what one 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 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 336 and in "Errors in commands" on page 357. 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 "How to run make" on page 409.

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

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 325.
- 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 425.
- 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 311).
- 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 318).
  - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see "Conditional parts of makefiles" on page 387).
  - Defining a variable from a verbatim string containing multiple lines (see "Defining variables verbatim" on page 384).
- 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 427.

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 make-file 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 369.

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 380) or pattern rules (see "Defining and redefining pattern rules" on page 438).

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 348.

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 options for make" on page 419). Then the directories (if they exist) are searched, in the following order.

1. '*`prefix`***`/include`**' (normally, '**`/usr/local/include`**')
2. '**`/usr/gnu/include`**',
3. '**`/usr/local/include`**', '**`/usr/include`**'.

If an included makefile cannot be found in any of these directories, a warning message is generated, but it is not an immediately fatal error; processing of the makefile containing the **`include`** continues. Once it has finished reading makefiles, **`make`** will try to remake any that are out of date or don't exist. See "How makefiles are remade" on page 321. 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 variable, `MAKEFILES`

If the environment variable, `MAKEFILES`, is defined, **`make`** considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive in that various directories are searched for those files (see “Including other makefiles” on page 318). 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 make” on page 360). 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 334).

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 318.

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 427, 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 347). 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 317. 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 317) 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 413), 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 438 for more information on pattern rules.

For example, if you have a makefile called 'Makefile' that says how to make the target 'foo' (and other targets), you can write a makefile called 'GNUmakefile' that contains the following.

```
foo:
        frobnicate > foo

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

If you say 'make foo', make will find 'GNUmakefile', read it, and see that to make 'foo', it needs to run the command 'frobnicate > foo'. If you say 'make bar', **make** will find no way to make 'bar' in'GNUmakefile', so it will use the commands from the pattern rule: 'make -f Makefile bar'. If 'Makefile' provides a rule for updating 'bar', **make** will apply the rule. And likewise for any other target that 'GNUmakefile' does not say how to make.

The way this works is that the pattern rule has a pattern of just '%', so it matches any target whatever. The rule specifies a dependency 'force', to guarantee that the commands will be run even if the target file already exists. We give 'force' target empty commands to prevent **make** from searching for an implicit rule to build it—otherwise it would apply the same match-anything rule to 'force' itself and create a dependency loop!

4

Writing rules

A *rule* appears in the makefile and says when and how to remake certain files, called the rule's *targets* (most often only one per rule). It lists the other files that are the *dependencies* of the target, and *commands* to use to create or update the target. The following documentation discusses the general rules for makefiles.

- “Generating dependencies automatically” on page 348

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 438.

Therefore, we usually write the makefile so that the first rule is the one for compiling the entire program or all the programs described by the makefile (often with a target called ‘`all`’). See “Arguments to specify the goals” on page 411.

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 328) and a name of the form ‘*a*(*m*)’ represents member *m* in archive file *a* (see “Archive members as targets” on page 452). 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 342). 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 351.

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 369). 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 451) 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 infor-mation 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 351).

Using wildcard characters in file names

A single file name can specify many files using *wildcard characters*. The wildcard characters in make are '`*`', '`?`' and '`[...]`', the same as in the Bourne shell. For example, '`*.c`' specifies a list of all the files (in the working directory) whose names end in '`.c`'.

The character '`~`' at the beginning of a file name also has special significance. If alone, or followed by a slash, it represents your home directory. For example '`~/bin`' expands to '`/home/you/bin`'. If the '`~`' is followed by a word, the string represents the home directory of the user named by that word. For example '`~john/bin`' expands to '`/home/john/bin`'.

Wildcard expansion happens automatically in targets, in dependencies, and in commands (where the shell does the expansion). In other contexts, wildcard expansion happens only if you request it explicitly with the `wildcard` function. The special significance of a wildcard character can be turned off by preceding it with a backslash. Thus, '`foo\*bar`' would refer to a specific file whose name consists of '`foo`', an asterisk, and '`bar`'.

Wildcard examples

Wildcards can be used in the commands of a rule, where they are expanded by the shell. For example, here is a rule to delete all the object files:

```
clean:
        rm -f *.o
```

Wildcards are also useful in the dependencies of a rule. With the following rule in the makefile, '`make print`' will print all the '`.c`' files that have changed since the last time you printed them:

```
print: *.c
        lpr -p $?
        touch print
```

This rule uses '`print`' as an empty target file; see "Empty target files to record events" on page 339. (The '`$?`' automatic variable is used to print only those files that have changed; see "Automatic variables" on page 440.) 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 331.

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 331.

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 330).

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 397.

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 372 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 di-rectory 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.

`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 333. In the `VPATH` variable, directory names are separated by colons or blanks. The order in which directories are listed is the order followed by `make` in its search. For example, `VPATH = src:../headers` specifies a path containing two directories, ‘`src`’ and ‘`../headers`’, which make searches in that order. With this value of `VPATH`, the rule, `foo.o : foo.c` is interpreted as if it were written: `foo.o : src/foo.c`.

This is assuming the file, ‘`foo.c`’, does not exist in the current directory but is found in the directory, ‘`src`’.

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 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 438). 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
```

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 440).

4: Writing rules

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 434.

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 427).

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, '`-l`*name*'. (You can tell something strange is going on here because the dependency is normally the name of a file, and the *file name* of the library looks like '`lib name.a`', not like '`-l`*name*'.)

When a dependency's name has the form '`-l`*name*', `make` handles it specially by searching for the file '`lib`*name*`.a`' in the current directory, in directories specified by matching `vpath` search paths and the `VPATH` search path, and then in the directories '`/lib`', '`/usr/lib`', and '*prefix*`/lib`' (normally, '`/usr/local/lib`'). 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 340), 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 425). 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 411).

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
```

4: Writing rules

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 336.

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 440).

Special built-in target names

Certain names have special meanings if they appear as targets.

`.PHONY`

The dependencies of the special target, `.PHONY`, are considered to be phony targets. When it is time to consider such a target, `make` will run its commands unconditionally, regardless of whether a file with that name exists or what its last-modification time is. See “Phony targets” on page 336.

`.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 446.

`.DEFAULT`

The commands specified for `.DEFAULT` are used for any tar-get for which no rules are found (either explicit rules or im-plicit rules). See “Defining last-resort default rules” on page 445. 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 448.

`.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 make” on page 359. 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 437.

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 357.

`.SILENT`

If you specify dependencies for `.SILENT`, then make will not the print commands to remake those particular files before executing them. The commands for `.SILENT` are not meaningful.

If mentioned as a target with no dependencies, `.SILENT` says not to print any commands before executing them. This usage of '`.SILENT`' is supported only for historical compatibility. We recommend you use the more selective ways to silence specific commands. See "Command echoing" on page 353.

If you want to silence all commands for a particular run of `make`, use the '`-s`' or '`--silent`' options. See "Summary of options for make" on page 419.

`.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" on page 361.

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 446.

Multiple targets in a rule

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

This is 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 440). 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
  ```

  Here we assume 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 397 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 344.

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 wrinkle 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 416). 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 427).

Static pattern rules

Static pattern rules are rules which specify multiple targets and con-struct the dependency names for each target based on the target name. They are more general than ordinary rules with multiple targets be-cause 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 328).

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 back-slashes ('\'). 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

$(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 440.

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 397), 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 versus 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 438). Both have a pattern for the target and patterns for constructing the names of dependencies. The difference is in how `make` decides *when* the rule applies.

An implicit rule can apply to any target that matches its pattern, but it *does* apply only when the target has no commands otherwise specified, and only when the dependencies can be found. If more than one implicit rule appears applicable, only one applies; the choice depends on the order of rules.

By contrast, a static pattern rule applies to the precise list of targets that you specify in the rule. It cannot apply to any other target and it invariably does apply to each of the targets specified. If two conflicting rules apply, and both have commands, that’s an error. The static pattern rule can be better than an implicit rule for 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.

The double-colon rules for a target are executed in the order they appear in the makefile. However, the cases where double-colon rules really make sense are those where the order of executing the commands would not matter.

Double-colon rules are somewhat obscure and not often very useful; they provide a mechanism for cases in which the method used to update a target differs depending on which dependency files caused the update, and such cases are rare.

Each double-colon rule should specify commands; if it does not, an implicit rule will be used if one applies. See "Using implicit rules" on page 427.

Generating dependencies automatically

In the makefile for a program, many of the rules you need to write often say only that some object file depends on some header file. For example, if ‘`main.c`’ uses ‘`defs.h`’ via an `#include`, you would write: `main.o: defs.h`.

You need this rule so that make knows that it must remake ‘`main.o`’ whenever ‘`defs.h`’ changes. You can see that for a large program you would have to write dozens of such rules in your makefile. And, you must always be very careful to update the makefile every time you add or remove an `#include`.

To avoid this hassle, most modern C compilers can write these rules for you, by looking at the `#include` lines in the source files.

Usually this is done with the ‘`-M`’ option to the compiler. For example, the command, `cc -M main.c`, generates the output: `main.o : main.c defs.h`.

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

NOTE: Such a dependency constitutes mentioning ‘`main.o`’ in a makefile, so it can never be considered an intermediate file by implicit rule search. This means that `make` won’t ever remove the file after using it; see “Chains of implicit rules” on page 437.

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 318).

In GNU `make`, the feature of remaking makefiles makes this practice obsolete—you need never tell `make` explicitly to regenerate the dependencies, because it always regenerates any makefile that is out of date. See “How makefiles are remade” on page 321.

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 (i.e., a makefile) called ‘*`name`*`.d`’ from a C source file called ‘*`name`*`.c`’.

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

See “Defining and redefining pattern rules” on page 438 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’ve 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 318. 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 375 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 321.

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.

Each command line must start with a tab, except that the first command line may be attached to the *target-and-dependencies* line with a semicolon in between. Blank lines and lines of just comments may appear among the command lines; they are ignored. (But beware, an apparently "blank" line that begins with a tab is *not* blank! It is an empty command; see "Using empty commands" on page 367.)

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 354.

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 ‘`-n`’ or ‘`--just-print`’, echoing is all that happens with no execution. See “Summary of options for make” on page 419. 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 340). `.SILENT` is essentially obsolete since ‘`@`’ is more flexible.

Command execution

When it is time to execute commands to update a target, they are executed by making a new subshell for each line. (In practice, `make` may take shortcuts that do not affect the results.)

NOTE: 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.

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 385.

Parallel execution

GNU `make` knows how to execute several commands at once. Normally, `make` will execute only one command at a time, waiting for it to finish before executing the next. However, the '`-j`' or '`--jobs`' option tells `make` to execute many commands simultaneously.

If the '`-j`' option is followed by an integer, this is the number of commands to execute at once; this is called the number of *job slots*.If there is nothing looking like an integer after the '`-j`' option, there is no limit on the number of job slots. The default number of job slots is one which means serial execution (one thing at a time).

One unpleasant consequence of running several commands simultaneously is that output from all of the commands comes when the commands send it, so messages from different commands may be interspersed.

Another problem is that two processes cannot both take input from the same device; so to make sure that only one command tries to take input from the terminal at once, `make` will invalidate the standard input streams of all but one running command. This means that attempting to read from standard input will usually be a fatal error (a '`Broken pipe`' signal) for most child processes if there are several.

It is unpredictable which command will have a valid standard input stream (which will come from the terminal, or wherever you redirect the standard input of `make`). The first command run will always get it first, and the first command started after that one finishes will get it next, and so on.

We will change how this aspect of `make` works if we find a better alternative. In the mean time, you should *not* rely on any command using standard input at all if you are using the parallel execution feature; but if you are *not* using this feature, then standard input works normally in all commands.

If a command fails (i.e., 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 357), 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 options for make" on page 419), `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 the '-i' flag, make treats an error return just like success, except that it prints out a message that tells you the status code the command exited with, and says that the error has been ignored.

When an error happens that make has not been told to ignore, it implies that the current target cannot be correctly remade, and neither can any other that depends on it either directly or indirectly. No further commands will be executed for these targets, since their preconditions have not been achieved.

Normally make gives up immediately in this circumstance, returning a nonzero status. However, if the '-k' or '--keep-going' flag is specified, make continues to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, 'make -k' will continue compiling other object files even though it already knows that linking them will be impossible. See "Summary of options for make" on page 419.

The usual behavior assumes that your purpose is to get the specified targets up to date; once make learns that this is impossible, it might as well report the failure immediately. The '-k' option says that the real purpose is to test as many of the changes made in the program as possible, perhaps to find several independent problems so that you can correct them all before the next attempt to compile.

This is why Emacs' `compile` command passes the '-k' flag by default.

Usually when a command fails, if it has changed the target file at all, the file is corrupted and cannot be used—or at least it is not completely updated. Yet the file's timestamp says that it is now up to date, so the next time `make` runs, it will not try to update that file. The situation is just the same as when the command is killed by a signal; see "Interrupting or killing make" on page 359. 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 make

If make gets a fatal signal while a command is executing, it may delete the target file that the command was supposed to update. This is done if the target file's last-modification time has changed since make first checked it.

The purpose of deleting the target is to make sure that it is remade from scratch when make is next run. Why is this? Suppose you 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 make

Recursive use of make means using make as a command in a makefile. This technique is useful when you want separate makefiles for various subsystems that compose a larger system. For example, suppose you have a subdirectory, 'subdir', which has its own makefile, and you would like the containing directory's makefile to run make on the subdirectory. You can do it by writing the following

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

Or, equivalently (see "Summary of options for make" on page 419), 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.

How the MAKE variable works

Recursive make commands should always use the variable, MAKE, not the explicit command name, 'make', as shown in the following example.

```
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 413.

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 413. 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 com-mand line of a rule contains the variable MAKE, the flags '-t', '-n' and '-q' do not apply to that line.

Command lines containing `MAKE` are exe-cuted 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” on page Communicating options to a sub-make), so your request to touch the files, or print the commands, is propagated to the subsystem.

Communicating variables to a sub-make

Variable values of the top-level make can be passed to the sub-make through the environment by explicit request. These variables are defined in the sub-make as defaults, but do not override what is specified in the makefile used by the sub-`make` makefile unless you use the ‘`-e`’ switch (see “Summary of options for make” on page 419).

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 385.

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” on page 362.

Variables are *not* normally passed down if they were created by default by `make` (see “Variables used by implicit rules” on page 434). 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 381. You may notice that the `export` and `unexport` directives work in make in the same way they work in the shell, `sh`.

If you want all variables to be exported by default, you can use export by itself: `export`. This tells `make` that variables which are not explicitly mentioned in an `export` or `unexport` directive should be exported. Any variable given in an `unexport` directive will still not be exported. If you use `export` by itself to export variables by default, variables whose names contain characters other than alphanumerics and underscores will not be exported unless specifically mentioned in an `export` directive.

The behavior elicited by an `export` directive by itself was the default in older versions of GNU `make`. If your makefiles depend on this behavior and you want to be compatible with old versions of `make`, you can write a rule for the special target, `.EXPORT_ALL_VARIABLES`, instead of using the `export` directive. This will be ignored by old `make`s, while the `export` directive will cause a syntax error.

Likewise, you can use `unexport` by itself to tell make not to export variables by default. Since this is the default behavior, you would only need to do this if export had been used by itself earlier (in an included makefile, perhaps). You ***cannot*** use `export` and `unexport` by themselves to have variables exported for some commands and not for others. The last `export` or `unexport` directive that appears by itself determines the behavior for the entire run of `make`.

As a special feature, the variable, `MAKELEVEL`, is changed when it is passed down from level to level. This variable’s value is a string which is the depth of the level as a decimal number. The value is ‘`0`’ for the top-level make;‘`1`’ for a sub-`make`,‘`2`’ for a sub-sub-`make`, and so on. The incrementation happens when `make` sets up the environment for a command.

The main use of `MAKELEVEL` is to test it in a conditional directive (see “Conditional parts of makefiles” on page 387); 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 variable, MAKEFILES” on page 320.

Communicating options to a sub-`make`

Flags such as ‘`-s`’ and ‘`-k`’ are passed automatically to the sub-`make` through the variable, `MAKEFLAGS`. This variable is set up automatically by `make` to contain the flag letters that `make` received. Thus, if you do ‘`make -ks`’ then `MAKEFLAGS` gets the value 'ks'.

As a consequence, every sub-make gets a value for MAKEFLAGS in its environment. In response, it takes the flags from that value and processes them as if they had been given as arguments. See "Summary of options for make" on page 419.

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 416.

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 355). 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 options for make" on page 419), set each time you run `make`. You simply put a value for `MAKEFLAGS` in your environment. You can also set `MAKEFLAGS` in a makefile, to specify additional flags that should also be in effect for that makefile.

NOTE: 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 '`-C`' option, and in sub-`make`s. `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 384 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 371). 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 427).

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 options for make" on page 419.

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 $@
endef
```

`make` will not echo the first line, the `echo` command. But it will echo the following two command lines. On the other hand, prefix characters on the command line that refers to a canned sequence apply to every line in the sequence. So the 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.

You may be wondering why you would want to define a command string that does nothing. The only reason this is useful is to prevent a target from getting implicit commands (from implicit rules or the `.DEFAULT` special target; see “Implicit rules” on page 425 and “Defining last-resort default rules” on page 445).

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 336 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.

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” on page 361).

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 416).

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 440.

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 440).

The two flavors of variables

There are two ways that a variable in GNU make can have a value; we call them the two *flavors* of variables. The two flavors are distinguished in how they are defined and in what they do when expanded.

The first flavor of variable is a *recursively expanded* variable. Variables of this sort are defined by lines using ‘=’ (see “Setting variables” on page 380) or by the define directive (see “Defining variables verbatim” on page 384). 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 ex-panding 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 395).

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 380). 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
```

This 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 407. 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” on page 361 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 395).

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!)

Advanced features for reference to variables

The following discussion describes some advanced features that you can use to reference variables in more flexible ways.

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. Consider the following statement.

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

This input sets '`bar`' to'**`a.c b.c c.c`**'. See "Setting variables" on page 380. A substitution reference is actually an abbreviation for use of the `patsubst` expansion function; see "Functions for string substitution and analysis" on page 397. 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 397 for a description of the `patsubst` function. Consider the following example.

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

This statement sets '`bar`' to'**`a.c b.c c.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, read on.

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 395), just like any other reference. For instance, the following example uses the `subst` function (see "Functions for string substitution and analysis" on page 397):

```
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
1_dirs := dir1 dir2

a_files := filea fileb
1_files := file1 file2

ifeq “$(use_a)” “yes”
a1 := a
else
a1 := 1
endif

ifeq “$(use_dirs)” “yes”
df := dirs
else
df := files
endif

dirs := $($(a1)_$(df))
```

The previous example will give `dirs` the same value as `a_dirs`, `1_dirs`, `a_files` or `1_files` depending on the settings of `use_a` and `use_dirs`.

Computed variable names can also be used in substitution references:

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

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

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 bg 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 db gq 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`'.

NOTE: *Nested variable references* are quite different from *recursively expanded variables* (see "The two flavors of variables" on page 372), 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 an overriding value when you run `make`. See “Overriding variables” on page 416.
- You can specify a value in the makefile, either with an assignment (see “Setting variables” on page 380) or with a verbatim definition (see “Defining variables verbatim” on page 384).
- Variables in the environment become `make` variables. See “Variables from the environment” on page 385.
- Several automatic variables are given new values for each rule. Each of these has a single conventional use. See “Automatic variables” on page 440.
- Several variables have constant initial values. See “Variables used by implicit rules” on page 434.

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 372.

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 434). Several special variables are set automatically to a new value for each rule; these are called the *automatic variables* (see "Automatic variables" on page 440).

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 '=': it defines a recursively-expanded variable. However, when there is a previous definition, exactly what '+=' does depends on what flavor of variable you defined originally. See "The two flavors of variables" on page 372 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 380 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 372). 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 429.) 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 416), 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 381. The `override` directive was not invented for escalation in the war between makefiles and command arguments. It was invented so you can alter and add to values that the user specifies with command arguments.

For example, suppose you always want the ‘`-g`’ switch when you run the C compiler, but you would like to allow the user to specify the other switches with a command argument just as usual. You could use this override directive: `override CFLAGS += -g`.

You can also use `override` directives with `define` directives. This is done as you might expect, as in the following.

```
override define foo
bar
endef
```

See “Defining variables verbatim” on page 384 for information about `define`.

6: How to use variables

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 365).

The `define` directive is followed on the same line by the name of the variable and nothing more. The value to give the variable appears on the following lines. The end of the value is marked by a line containing just the word, `endef`. Aside from this difference in syntax, `define` works just like ‘`=`’; it creates a recursively-expanded variable (see “The two flavors of variables” on page 372). The variable name may contain function and variable references which are expanded when the directive is read to find the actual variable name to use.

```
define two-lines
echo foo
echo $(bar)
endef
```

The value in an ordinary assignment cannot contain a newline; the newlines that separate the lines of the value in a `define` become part of the variable’s value (except for the final newline which precedes the `endef` and is not considered part of the value). The previous example is functionally equivalent to `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 354.

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)
endef
```

See “The override directive” on page 383.

Variables from the environment

Variables in **make** can come from the environment in which **make** is run. Every environment variable that **make** sees when it starts up is transformed into a **make** variable with the same name and value. But an explicit assignment in the makefile, or with a command argument, overrides the environment. If the '-e' flag is specified, then values from the environment override assignments in the makefile (see "Summary of options for make" on page 419), although this is not arecommended 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 make" on page 360). 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" on page 361 for full details.

Other use of variables from the environment is not recommended. It is not wise for makefiles to depend for their functioning on environment variables set up outside their control, since this would cause different users to get different results from the same makefile. This is against the whole purpose of most makefiles.

Such problems would be especially likely with the variable, SHELL, which is normally present in the environment to specify the user's choice of interactive shell. It would be very undesirable for this choice to affect **make**. So **make** ignores the environment value of SHELL.

7

Conditional parts of makefiles

A *conditional* causes part of a makefile to be obeyed or ignored depending on the values of variables. Conditionals can compare the value of one variable to another, or the value of a variable to a constant string. Conditionals control what **make** actually "sees" in the makefile, so they cannot be used to control shell commands at the time of execution. The following documentation describes conditionals in more detail.

- "Example of a conditional" on page 388
- "Syntax of conditionals" on page 390
- "Conditionals that test flags" on page 393

Example of a conditional

The following example of a conditional tells **make** to use one set of libraries if the CC variable is ‘gcc’, and a different set of libraries otherwise. It works by controlling which of two command lines will be used as the command for a rule. The result is that ‘CC=gcc’ as an argument to **make** changes not only which compiler is used but also which libraries are linked.

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

This conditional uses three directives: one `ifeq`, one `else` and one `endif`.

The `ifeq` directive begins the conditional, 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. This is why the larger syntactic units of the makefile, such as rules, may cross the beginning or the end of the conditional.

When the variable, CC, has the value ‘gcc’, the 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 397) 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, the *`text-if-false`*, if any, is effective.

`ifdef `*`variable-name`*

If the variable *`variable-name`* has a non-empty value, the *`text-if-true`* is effective; otherwise, the *`text-if-false`*, if any, is effective. Variables that have never been defined have an empty value.

NOTE: `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 440). 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 397). 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 make" on page 360.

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.

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 string 'fEEt on the strEEt'.

`$(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 375.

```
$(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 387).

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 393for 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 332). 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 383).

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 that the other, the extra words are copied unchanged into the result.

For example, '`$(join a b,.c .o)`' produces '**`a.c b.o`**'.

Whitespace between the words in the lists is not preserved; it is replaced with a single space. This function can merge the results of the `dir` and `notdir` functions to produce the original list of files which was given to those two functions.

`$(word n, text)`

Returns the *n*th word of *text*. The legitimate values of *n* start from 1. If *n* is bigger than the number of words in *text*, the value is empty.

For example, `$(word 2, foo bar baz)` returns '**`bar`**'.

`$(words text)`

Returns the number of words in *text*. Thus, the last word of *text* is `$(word $(words text),text)`.

`$(firstword names...)`

The argument, *names*, is regarded as a series of names, separated by whitespace. The value is the first name in the series. The rest of the names are ignored.

For example, `$(firstword foo bar)` produces the result '**`foo`**'.

Although `$(firstword text)` is the same as `$(word 1, text)`, the `firstword` function is retained for its simplicity.

`$(wildcard pattern)`

The argument *`pattern`* is a file name pattern, typically containing wildcard characters (as in shell file name patterns). The result of wildcard is a space-separated list of the names of existing files that match the pattern. See “Using wildcard characters in file names” on page 328.

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 sub-stitution 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 372.

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!`’ (es un nombre bastante largo, no?), 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)
```

NOTE: *`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 434.

NOTE: 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 options for make" on page 419).

'`environment override`'
: Used if *`variable`* was defined as an environment variable and the '`-e`' option is turned on (see "Summary of options for make" on page 419).

'`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 383).

'`automatic`'
: Used if *`variable`* is an automatic variable defined for the execution of the commands for each rule (see "Automatic variables" on page 440).

This information is primarily useful (other than for your curiosity) to determine if you want to believe the value of a variable. For example, suppose you have a makefile, ‘`foo`’, that includes another makefile, ‘`bar`’.

You want a variable, `bletch`, to be defined in ‘`bar`’ if you run the command, ‘`make -f bar`’, even if the environment contains a definition of `bletch`. However, if ‘`foo`’ defined `bletch` before including ‘`bar`’, you do not want to override that definition. This could be done by using an `override` directive in ‘`foo`’, giving that definition precedence over the later definition in ‘`bar`’; unfortunately, the `override` directive would also override any command line definitions. So, ‘`bar`’ could include the following.

```
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.

```
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 397.

The `shell` function

The `shell` function is unlike any other function except the `wildcard` function (see “The wildcard function” on page 331) in that it communicates with the world outside of **`make`**.

The `shell` function performs the same function that backquotes (‘`‘`’) perform in most shells: it does *command expansion*. This means that it takes an argument that is a shell command and returns the output of the command. The only processing **`make`** does on the result, before substituting it into the surrounding text, is to convert newlines to spaces.

The commands run by calls to the `shell` function are run when the function calls are expanded. In most cases, this is when the makefile is read in. The exception is that function calls in the commands of the rules are expanded when the commands are run, and this applies to `shell` function calls like all others. 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 `make`

A makefile that says how to recompile a program can be used in more than one way. The simplest use is to recompile every file that is out of date.

Usually, makefiles are written so that if you run `make` with no arguments, it does just that. 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

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 413.

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 315).

Arguments to specify the goals

The *goals* are the targets that `make` should strive ultimately to update. Other targets are updated as well if they appear as dependencies of goals, or dependencies of dependencies of goals, etc.

By default, the goal is the first target in the makefile (not counting targets that start with a period). Therefore, makefiles are usually written so that the first target is for compiling the entire program or programs they describe. If the first rule in the makefile has several targets, only the first target in the rule becomes the default goal, not the whole list.

You can specify a different goal or goals with arguments to `make`. Use the name of the goal as an argument. If you specify several goals, make processes each of them in turn, in the order you name them. Any target in the makefile may be specified as a goal (unless it starts with '`-`' or contains an '`=`', in which case it will be parsed as a switch or variable definition, respectively). Even targets not in the makefile may be specified, if `make` can find implicit rules that say how to make them.

One use of specifying a goal is if you want to compile only a part of the program, or only one of several programs. Specify as a goal each file that you wish to remake. For example, consider a directory containing several programs, with a makefile that starts like 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 336) or empty target (see "Empty target files to record events" on page 339). 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 468 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 360.

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 com-mands, 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 options for make” on page 419).

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 options for make" on page 419), 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 434 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 383).

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 options for make" on page 419).

The usual behavior of `make` assumes that your purpose is to get the goals up to date; once make learns that this is impossible, it might as well report the failure immediately. The '`-k`' flag says that the real purpose is to test as much as possible of the changes made in the program, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs' Meta-`x compile` command passes the '`-k`' flag by default.

9: How to run make

10

Summary of options for `make`

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 make" on page 360).

'`-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 385.

‘-f *file*’
‘--file=*file*’
‘--makefile=*file*’
Read the file named *file* as a makefile. See “Writing makefiles” on page 315.

‘-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 357.

‘-I *dir*’
‘--include-dir=*dir*’
Specifies a directory, *dir*, to search for included makefiles. See “Including other makefiles” on page 318. 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 355 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 417.

‘-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 355.

'-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 413.

'-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 415.

'-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 422). To print the data base without tryimng 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 413.

'-r'
'--no-builtin-rules'

Eliminate use of the built-in implicit rules (see "Using implicit rules" on page 427). You can still define your own by writing pattern rules (see "Defining and redefining pattern rules" on page 438). The '-r' option also clears out the default list of suffixes for suffix rules (see "Old-fashioned suffix rules" on page 446). But you can still define your own suffixes with a rule for .SUFFIXES, and then define your own suffix rules.

'-s'
'--silent'
'--quiet'

Silent operation; do not print the commands as they are executed. See "Command echoing" on page 353.

'-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 make" on page 360).

'-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 413.

'-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 make" on page 360. (In practice, you rarely need to specify this option since 'make' does it for you; see "The '--print-directory' option" on page 364.)

'--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 364.

'-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 413.

'`--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 the GNU C compiler, **`gcc`**. The following documentation describes in more detail the rules of remaking target files.

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 437.

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 434.

You can define your own implicit rules by writing *pattern rules*. See “Defining and redefining pattern rules” on page 438.

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 446.

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 429 for a catalogue of all the predefined implicit rules.

Previously, we said an implicit rule applies if the required dependencies “exist or can be made”. A file “can be made” if it is mentioned explicitly in the makefile as a target or a dependency, or if an implicit rule can be recursively found for how to make it. When an implicit dependency is the result of another implicit rule, we say that chaining is occurring. See “Chains of implicit rules” on page 437.

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 448 for the details of how the search is done.

NOTE: 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 429).

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 367).

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 444 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 (described in the following) 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 446 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 assembler, **as**. The precise command is:

```
$(AS) $(ASFLAGS)
```

'*n*.s' is made automatically from '*n*.S' by running the C preprocessor, **cpp**. The precise command is:

```
$(CPP) $(CPPFLAGS)
```

Linking a single object file

'*n*' is made automatically from '*n*.o' by running the linker (usually called **ld**) via the C compiler. The precise command used is:

```
$(CC) $(LDFLAGS) n.o $(LOADLIBES)
```

This rule does the right thing for a simple program with only one source file. It will also do the right thing if there are multiple object files (presumably coming from various other source files), one of which has a name matching that of the executable file.

Thus, `x: y.o z.o`, when 'x.c', 'y.c' and 'z.c' all exist will execute 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’s how it’s done.

Yacc for C programs

‘*n*.**c**’ is made automatically from ‘*n*.**y**’ by running Yacc with the command:

```
$(YACC) $(YFLAGS)
```

Lex for C programs

‘*n*.**c**’ is made automatically from ‘*n*.**l**’ by by running Lex. The actual command is:

```
$(LEX) $(LFLAGS)
```

Lex for Ratfor programs

‘*n*.**r**’ is made automatically from ‘*n*.**l**’ by by running Lex. The actual command is:

```
$(LEX) $(LFLAGS)
```

The convention of using the same suffix ‘**`.l`**’ for all Lex files regardless of whether they produce 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 443), 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 443), 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 434.

However, the commands in built-in implicit rules actually use variables such as `COMPILE.c`, `LINK.p`, and `PREPROCESS.S`, whose values contain the commands listed 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 332). 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 are variables used as names of programs in built-in rules.

`AR`
: Archive-maintaining program; default: '`ar`'.

`AS`
: Program for doing assembly; default: '`as`'.

`CC`
: Program for compiling C programs; default: '`cc`'.

`CXX`
: Program for compiling C++ programs; default: '`g++`'.

`CO`
: Program for extracting a file from RCS; default: '`co`'.

`CPP`
: Program for running the C preprocessor, with results to standard output; default: '`$(CC) -E`'.

`FC`
: Program for compiling or preprocessing Fortran and Ratfor programs; default: '`f77`'.

`GET`
: Program for extracting a file from SCCS; default: '`get`'.

`LEX`
: Program to use to turn Lex grammars into C programs or Ratfor programs; default: ‘`lex`’.

`PC`
: Program for compiling Pascal programs; default: ‘`pc`’.

`YACC`
: Program to use to turn Yacc grammars into C programs; default: ‘`yacc`’.

`YACCR`
: Program to use to turn Yacc grammars into Ratfor programs; default: ‘`yacc -r`’.

`MAKEINFO`
: Program to convert a Texinfo source file into an Info file; default: ‘`makeinfo`’.

`TEX`
: Program to make TEX DVI files from TEX source; default: ‘`tex`’.

`TEXI2DVI`
: Program to make TEX DVI files from Texinfo source; default: ‘`texi2dvi`’.

`WEAVE`
: Program to translate Web into TEX; default: ‘`weave`’.

`CWEAVE`
: Program to translate C Web into TEX; default: ‘`cweave`’.

`TANGLE`
: Program to translate Web into Pascal; default: ‘`tangle`’.

`CTANGLE`
: Program to translate C Web into C; default: ‘`ctangle`’.

`RM`
: Command to remove a file; default: ‘`rm -f`’.

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 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 make" on page 359.

A chain can involve more than two implicit rules. For example, it is possible to make a file 'foo' from 'RCS/foo.y,v' by running RCS, Yacc and **cc**. Then both 'foo.y' and 'foo.c' are intermediate files that are deleted at the end.

No single implicit rule can appear more than once in a chain. This means that **make** will not even consider such a ridiculous thing as making 'foo' from 'foo.o.o' by running the linker twice. This constraint has the added benefit of preventing any infinite loop in the search for an implicit rule chain.

There are some special implicit rules to optimize certain cases that would otherwise be handled by rule chains. For example, making 'foo' from 'foo.c' could be handled by compiling and linking with separate chained rules, using 'foo.o' as an intermediate file. But what actually happens is that a special rule for this case does the compilation and linking with a single cc command. The optimized rule is used in preference to the step-by-step chain because it comes earlier in the ordering of rules.

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`'.

NOTE: 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 369 and "Functions for transforming text" on page 395.

Introduction to pattern rules

A pattern rule contains the character '%' (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the '%' matches any nonempty substring, while other characters match only themselves.

For example, '`%.c`' as a pattern matches any file name that ends in '`.c`'. '`s.%.c`' as a pattern matches any file name that starts with '`s.`', ends in '`.c`' and is at least five characters long. (There must be at least one character to match the '%'.) The substring that the '%' matches is called the stem.

'%' in a dependency of a pattern rule stands for the same stem that was matched by the '%' in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration, and its dependency patterns must name files that exist or can be made. These files become dependencies of the target.

Thus, a rule of the 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 depen-dency 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 445.

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`**. First, the rule that compiles '**`.c`**' files into '**`.o`**' files:

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

defines a rule that can make any file '*`x`*`.o`' from '*`x`*`.c`'. The command uses the automatic variables '**`$@`**' and '**`$<`**' to substitute the names of the target file and the source file in each case where the rule applies (see "Automatic variables" on page 440).

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 443).

The following pattern rule has two targets:

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

This tells make that the command '`bison -d`*`x`*`.y`' will make both '*`x`*`.tab.c`' and '*`x`*`.tab.h`'. If the file '`foo`' depends on the files '`parse.tab.o`' and '`scan.o`' and the file '`scan.o`' depends on the file '`parse.tab.h`', when '`parse.y`' is changed, the command '`bison -d parse.y`' will be executed only once, and the dependencies of

11: Implicit rules

both ‘`parse.tab.o`’ and ‘`scan.o`’ will be satisfied. (Presumably the file ‘`parse.tab.o`’ will be recompiled from ‘`parse.tab.c`’ and the file ‘`scan.o`’ from ‘`scan.c`’, while ‘`foo`’ is linked from ‘`parse.tab.o`’, ‘`scan.o`’, and its other dependencies, and it will execute happily ever after.)

Automatic variables

Suppose you are writing a pattern rule to compile a ‘**`.c`**’ file into a ‘**`.o`**’ file: how do you write the ‘`cc`’ command so that it operates on the right source file name? You cannot write the name in the command, because the name is different each time the implicit rule is applied. What you do is use a special feature of **`make`**, the *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. By example, 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 ‘**`$@`**’ is the name of the archive file. In a pattern rule that has multiple targets (see “Introduction to pattern rules” on page 438), ‘**`$@`**’ 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 451. 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 427).

`$?`

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 451).

`$^`

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 451). 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 442). 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 '**$***' cannot be deter-mined in that way. Instead, if the target name ends with a recognized suffix (see "Old-fashioned suffix rules" on page 446), '**$***' 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 400).

NOTE: 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 452.

'$(<D)'
'$(<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 '`foo.c,v.o`' or from '`RCS/SCCS/s.foo.c,v`'. RCS and SCCS files are generally ultimate source files, which should not be remade from any other files; therefore, **`make`** can save time by not looking for ways to remake them.

If you do not mark the match-anything rule as terminal, then it is nonterminal. A 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 ‘`foo.c`’ matches the target for the pattern rule ‘`%.c : %.y`’ (the rule to run Yacc). Regardless of whether this rule is actually applicable (which happens only if there is a file ‘`foo.y`’), the fact that its target matches is enough to prevent consideration of any non-terminal match-anything rules for the file ‘`foo.c`’. Thus, **`make`** will not even consider trying to make ‘`foo.c`’ as an executable file from ‘`foo.c.o`’, ‘`foo.c.c`’, ‘`foo.c.p`’, etc.

The motivation for this constraint is that nonterminal match-anything rules are used for making files containing specific types of data (such as executable files) and a file name with a recognized suffix indicates some other specific type of data (such as a C source file).

Special built-in dummy pattern rules are provided solely to recognize certain file names so that nonterminal match-anything rules will not be considered. These dummy rules have no dependencies and no commands, and they are ignored for all other purposes. For example, the built-in implicit rule, `%.p :`, exists to make sure that Pascal source files such as ‘`foo.p`’ match a specific target pattern and thereby prevent time from being wasted looking for ‘`foo.p.o`’ or ‘`foo.p.c`’.

Dummy pattern rules such as the one for ‘`%.p`’ are made for every suffix listed as valid for use in suffix rules (see “Old-fashioned suffix rules” on page 446).

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 443). 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 367).

You can use a last-resort rule to override part of another makefile (see “Overriding part of another makefile” on page 323).

Old-fashioned suffix rules

Suffix rules are the old-fashioned way of defining implicit rules for **`make`**. Suffix rules are obsolete because pattern rules are more general and clearer. They are supported in GNU **`make`** for compatibility with old makefiles. They come in two kinds: *double-suffix* and *single-suffix*.

A double-suffix rule is defined by a pair of suffixes: the target suffix and the source suffix. It matches any file whose name ends with the target suffix. The corresponding implicit dependency is made by replacing the target suffix with the source suffix in the file name.

A two-suffix rule (whose target and source suffixes are '**`.o`**' and '**`.c`**') is equivalent to the pattern rule, '`%.o : %.c`'.

A single-suffix rule is defined by a single suffix, which is the source suffix. It matches any file name, and the corresponding implicit depen-dency 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 444). 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*)', the following algorithm is run twice, first using the entire target name, *t*, and, second, using '(*member*)' as the target, *t*, if the first run found no rule.

1. Split *t* into a directory part, called *d*, and the rest, called *n*. For example, if *t* is '`src/foo.o`', then *d* is '`src/`' and *n* is '`foo.o`'.
2. Make a list of all the pattern rules one of whose targets matches *t* or *n*. If the target pattern contains a slash, it is matched against *t*; otherwise, against *n*.
3. If any rule in that list is not a match-anything rule, then remove all 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.
6. If no pattern rule has been found so far, try harder. For each pattern rule in the list:
   a. If the rule is terminal, ignore it and go on to the next rule.
   b. Compute the dependency names as before.
   c. Test whether all the dependencies exist or ought to exist.
   d. For each dependency that does not exist, follow this algorithm recursively to see if the dependency can be made by an implicit rule.

   e. If all dependencies exist, ought to exist, or can be made by implicit rules, then this rule applies.

7. If no implicit rule applies, the rule for `.DEFAULT`, if any, applies. In that case, give *t* the same commands that `.DEFAULT` has. Otherwise, there are no commands for *t*.

Once a rule that applies has been found, for each target pattern of the rule other than the one that matched *t* or *n*, the '`%`' in the pattern is replaced with *s* and the resultant file name is stored until the commands to remake the target file, *t*, are executed. After these commands are executed, each of these stored file names are entered into the 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 440.

12

Using `make` to update archive files

Archive files are files containing named subfiles called *members*; they are maintained with the program, `ar`, and their main use is as subroutine libraries for linking.

The following documentation discusses `make`'s updating of your archive files.

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.

NOTE: 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 328. 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 named 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 '`(`' and '`)`' from being interpreted specially by the shell) in the presence of a file '`bar.c`' is enough to cause the following commands to be run, even without a makefile:

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

Here **`make`** has envisioned the '`bar.o`' as an intermediate file. See "Chains of implicit rules" on page 437.

Implicit rules such as this one are written using the automatic variable '`$%`'. See "Automatic variables" on page 440.

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 453 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 355) 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 446 for a full explanation of suffix rules.

Archive suffix rules are obsolete in GNU **make** because pattern rules for archives are a more general mechanism (see “Implicit rule for archive member targets” on page 453 for more information). But they are retained for compatibility with other **make**s.

To write a suffix rule for archives, you simply write a suffix rule using the target suffix, ‘**.a**’ (the usual suffix for archive files).

For example, the following is the old-fashioned suffix rule to update a library archive from C source files:

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

This works just as if you had written the 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 446). Thus a double-suffix rule, ‘.*x*.**a**’, produces two pattern rules: ‘(**%**.o): **%**.*x*’ and ‘**%.a**: **%**.*x*’.

13

Features of GNU `make`

The following summary compares the features of GNU `make` with other versions of `make`.

We consider the features of `make` in 4.2 BSD systems as a baseline. If you are concerned with writing portable makefiles, consider the following features of `make`, using them with caution; see also "Incompatibilities and missing features" on page 461. 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 332.
- Included makefiles. See "Including other makefiles" on page 318. 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 385.
- Options passed through the variable `MAKEFLAGS` to recursive invocations of `make`. See "Communicating options to a sub-make" on page 362.
- The automatic variable, **`$%`**, is set to the member name in an archive reference. See "Automatic variables" on page 440.

- 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 440.
- Substitution variable references. See "Basics of variable references" on page 371.
- 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 make" on page 360.
- Support for suffix '`.a`' in suffix rules. See "Suffix Rules for Archive Files" on page . This feature is obsolete in GNU `make` because the general feature of rule chaining (see "Chains of Implicit Rules" on page) allows one pattern rule for installing members in an archive (see "Implicit Rule for Archive Member Targets" on page) to be sufficient.
- The arrangement of lines and backslash-newline combinations in commands is retained when the commands are printed, so they appear as they do in the makefile, except for the stripping of initial whitespace.

The following features were inspired by various other versions of `make`. In some cases it is unclear exactly which versions inspired others.

- Pattern rules using '**`%`**'. This has been implemented in several versions of `make`. We're not sure who invented it first, but it's been spread around a bit. See "Defining and Redefining Pattern Rules" on page .
- Rule chaining and implicit intermediate files. This was implemented by Stu Feldman in his version of `make` for AT&T Eighth Edition Research Unix, and later by Andrew Hume of AT&T Bell Labs in his **`mk`** program (where he terms it "transitive closure"). We do not really know if we got this from either of them or thought it up ourselves at the same time. See "Chains of Implicit Rules" on page .
- The automatic variable, **`$^`**, containing a list of all dependencies of the current target. We did not invent this, but we have no idea who did. See "Automatic variables" on page 440. The automatic variable, **`$+`**, is a simple extension of **`$^`**.
- The "what if " flag ('`-W`' in GNU `make`) was (as far as we know) invented by Andrew Hume in **`mk`**. See "Instead of Executing the Commands" on page .
- 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 .

- Modified variable references using pattern substitution come from SunOS 4. See “Basics of variable references” on page 371. 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) 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 381.
- 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 453.
- 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 372.
- Pass command-line variable assignments automatically through the variable, `MAKE`, to recursive `make` invocations. See “Recursive use of make” on page 360.
- Use the ‘`-C`’ or ‘`--directory`’ command option to change directory. See “Summary of options for make” on page 419.
- Make verbatim variable definitions with `define`. See “Defining variables verbatim” on page 384.
- 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 336.
- Manipulate text by calling functions. See “Functions for transforming text” on page 395.
- Use the ‘`-o`’ or ‘`--old-file`’ option to pretend a file’s modification-time is old. See “Avoiding recompilation of some files” on page 415.

- 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 387.
- Specify a search path for included makefiles. See “Including other makefiles” on page 318.
- Specify extra makefiles to read with an environment variable. See “The variable, MAKEFILES” on page 320.
- 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 334.
- Allow suffixes for suffix rules (see “Old-fashioned suffix rules” on page 446) 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 make” on page 360.
- Specify static pattern rules. See “Static pattern rules” on page 344.
- Provide selective `vpath` search. See “Searching directories for dependencies” on page 332.
- Provide computed variable references. See “Basics of variable references” on page 371.
- Update makefiles. See “How makefiles are remade” on page 321. 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 429.
- The built-in variable, ‘`MAKE_VERSION`’, gives the version number of `make`.

14

Incompatibilities and missing features

The following documentation describes some incompatibilities and missing features in GNU `make`. See also “Problems and bugs” on page 463. 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))`’ stands for a member of archive file file. The member is chosen, not by name, but by being an object file which defines the linker symbol, `entry`.

  This feature was not put into GNU `make` because of the non-modularity of putting knowledge into `make` of the internal format of archive file symbol tables. See “Updating archive symbol directories” on page 453.

- Suffixes (used in suffix rules) that end with the character ‘`~`’ have a special meaning to System V `make`; they refer to the SCCS file that corresponds to the file one would get without the ‘`~`’. For example, the suffix rule, ‘`.c~.o`’, would make the file, ‘`n.o`’, from the SCCS file, ‘`s.n.c`’. For complete coverage, a whole series of such suffix rules is required. See “Old-fashioned suffix rules” on page 446.

  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 437.

- In System V make, the string, ‘$$@’, has the strange meaning that, in the dependencies of a rule with multiple targets, it stands for the particular target that is being processed.

  This is not defined in GNU `make` because ‘$$’ should always stand for an ordinary ‘$’. It is possible to get this functionality through the use of static pattern rules (see “Static pattern rules” on page 344).

  The System V make rule: `$(`*`targets`*`): $$@.o lib.a` can be replaced with the GNU `make` static pattern rule: `$(`*`targets`*`): %: %.o lib.a`.
- In System V and 4.3 BSD `make`, files found by `VPATH` search (see “Searching directories for dependencies” on page 332) have their names changed inside command strings. We feel it is much cleaner to always use automatic variables and thus make this feature obsolete.
- In some Unix `make`s, the automatic variable, `$*`, appearing in the dependencies of a rule, has the amazingly strange “feature” of expanding to the full name of the target of that rule. We cannot imagine what went on in the minds of Unix `make` developers to do this; it is utterly inconsistent with the normal definition of `$*`.
- In some Unix `make`s, implicit rule search is apparently done for all targets, not just those without commands (see “Using implicit rules” on page 427). This means you can use: `foo.o: cc -c foo.c`, and Unix `make` will intuit that ‘`foo.o`’ depends on ‘`foo.c`’.

  We feel that such usage is broken. The dependency properties of `make` are well-defined (for GNU `make`, at least), and doing such a thing simply does not fit the model.
- GNU `make` does not include any built-in implicit rules for compiling or preprocessing EFL programs. If we hear of anyone who is using EFL, we will gladly add them.
- It appears that in SVR4 `make`, a suffix rule can be specified with no commands, and it is treated as if it had empty commands (see “Using empty commands” on page 367).

  For example, `.c.a:` will override the built-in ‘`.c.a`’ suffix rule. We feel that it is cleaner for a rule without commands to always simply add to the dependency list for the target. `.c.a:` can be easily rewritten to get the desired behavior in GNU `make: .c.a: ;`
- Some versions of `make` invoke the shell with the ‘`-e`’ flag, except under ‘`-k`’ (see “Testing the compilation of a program” on page 417). The ‘`-e`’ flag tells the shell to exit as soon as any program it runs returns a nonzero status.

  We feel it is cleaner to write each shell command line to stand on its own without requiring this special treatment.

Problems and bugs

If you have problems with GNU `make` or think you've found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you've actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible makefile that reproduces the problem. Then send us the makefile and the exact results make gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you've got a precise problem, please send electronic mail either through the Internet or via UUCP.

Internet address:

`bug-gnu-utils@prep.ai.mit.edu`

UUCP path:

`mit-eddie!prep.ai.mit.edu!bug-gnu-utils`

Please include the version number of `make` you are using. You can get this information with the command, `make --version`. Be sure also to include the type of machine and operating system you are using. If possible, include the contents of the file, `config.h`, generated by the configuration process.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or that are just obscure features, send a message to the bug reporting address. We cannot guarantee that you'll get help with your problem, but many seasoned `make` users read the mailing list and they will probably try to help you out. The maintainers sometimes answer such questions as well, when time permits.

15

Makefile conventions

The following discusses conventions for writing the Makefiles for GNU programs.

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. (This is never a problem with GNU `make`.)

```
SHELL = /bin/sh
```

Different `make` programs have incompatible suffix lists and implicit rules, and this sometimes creates confusion or misbehavior.

So it is a good idea to set the suffix list explicitly using only the suffixes you need in the particular Makefile, using something like the following.

```
.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 '`./`' and '`$(srcdir)/`' is important when using the '`--srcdir`' option to '`configure`'. A rule of the following form will fail when the current directory is not the source directory, because '`foo.man`' and '`sedscript`' are not in the current 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.

```
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.

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

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 echo egrep expr grep
ln mkdir mv pwd rm rmdir sed test touch
```

Stick to the generally supported options for these programs. For example, don't use '`mkdir -p`', convenient as it may be, because most systems don't support it. The Makefile rules for building and installation can also use compilers and related programs, but should do so using `make` variables so that the user can substitute alternatives.

The following are some of the programs.

```
ar bison cc flex install ld lex
make makeinfo ranlib texi2dvi yacc
```

Use the following `make` variables:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

When you use `ranlib`, you should make sure nothing bad happens if the system does not have `ranlib`. Arrange to ignore an error from that command, and print a message before the command to tell the user that failure of the `ranlib` command does not mean a problem.

If you use symbolic links, you should implement a fallback for systems that don't have symbolic links.

It is 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 tar-get. 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 472), 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`', be-cause 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 called '`mkinstalldirs`' which is convenient for this; find it in the Texinfo package.You can use a rule like this:

```
# Make sure all installation directories
# (e.g. $(bindir)) actually exist by
# making them if necessary.
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs       $(bindir) $(datadir) \
                                      $(libdir) $(infodir) \
                                      $(mandir)
```

This rule should not modify the directories where compilation is done. It should do nothing but create installation directories.

Variables for specifying commands

Makefiles should provide variables for overriding certain commands, options, and so on.

In particular, you should run most utility programs via variables. Thus, if you use Bison, have a variable named `BISON` whose default value is set with '`BISON = bison`', and refer to it with `\$(BISON)` whenever you need to use Bison.

File management utilities such as `ln`, `rm`, `mv`, and so on, need not be referred to through variables in this way, since users don't need to replace them with other programs.

Each program-name variable should come with an options variable that is used to supply options to the program. Append '`FLAGS`' to the program-name variable name to get the options variable name—for example, `BISONFLAGS`. (The name `CFLAGS` is an exception to this rule, but we keep it because it is standard.) Use `CPPFLAGS` in any compilation command that runs the preprocessor, and use `LDFLAGS` in any compilation command that does linking as well as in any direct use of ld.

If there are C compiler options that *must* be used for proper compilation of certain files, do not include them in `CFLAGS`. Users expect to be able to specify `CFLAGS` freely themselves. Instead, arrange to pass the necessary options to the C compiler independently of `CFLAGS`, by writing them explicitly in the compilation commands or by defining an implicit rule, like the following.

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

Do include the '`-g`' option in `CFLAGS` because that is not required for proper compilation. You can consider it a default that is only recommended. If the package is set up so that it is compiled with GCC by default, then you might as well include '`-O`' in the default value of `CFLAGS` as well. Put `CFLAGS` last in the compilation command, after other variables containing compiler options, so the user can use `CFLAGS` to override the others. Every Makefile should define the variable, `INSTALL`, which is the basic command for installing a file into the system. Every Makefile should also define the variables `INSTALL_PROGRAM` and `INSTALL_DATA`. (The default for each of these should be `\$(INSTALL)`.) Then it should use those variables as the commands for actual installation, for executables and nonexecutables respectively. Use these variables as follows:

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

Always use a file name, not a directory name, as the second argument of the installation commands. Use a separate command for each file to be installed.

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. 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 '`/`'.

'`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.

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`'.

'`sbindir`'

The directory for installing executable programs that can be run from the shell but are only generally useful to system administrators. This should normally be '`/usr/local/sbin`' but write it as '`\$(exec_prefix)/sbin`'.

'`libexecdir`'

The directory for installing executable programs to be run by other programs rather than by users. This directory should normally be '`/usr/local/libexec`', but write it as '`\$(exec_prefix)/libexec`'.

Data files used by the program during its execution are divided into categories in two ways.

- Some files are normally modified by programs; others are never normally modified (though users may edit some of these).
- Some files are architecture-independent and can be shared by all machines at a site; some are architecture-dependent and can be shared only by machines of the same kind and operating system; others may never be shared between two machines.

This makes for six different possibilities. However, we want to discourage the use of architecture-dependent files, aside from of object files and libraries. It is much cleaner to make other data files architecture-independent, and it is generally not hard.

Therefore, the following are the variables makefiles should use to specify directories.

‘`datadir`’

The directory for installing read-only architecture indepen-dent 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.

‘`sysconfdir`’

The directory for installing read-only data files that pertain to a single machine–that is to say, files for configuring a host. Mailer and network configuration files, ‘`/etc/passwd`’, and so forth belong here. All the files in this directory should be ordinary ASCII text files. This directory should normally be ‘`/usr/local/etc`’, but write it as ‘`\$(prefix)/etc`’.

Do not install executables in this directory (they probably belong in ‘`\$(libexecdir)`’ or ‘`\$(sbindir)`)’. Also do not install files that are modified in the normal course of their use (programs whose purpose is to change the configuration of the system excluded). Those probably belong in ‘`\$(localstatedir)`’.

‘`sharedstatedir`’

The directory for installing architecture-independent data files which the programs modify while they run. This should normally be ‘`/usr/local/com`’, but write it as ‘`\$(prefix)/com`’.

'`localstatedir`'
The directory for installing data files which the programs modify while they run, and that pertain to one specific machine. Users should never need to modify files in this directory to configure the package's operation; put such configu-ration information in separate files that go in '`datadir`' or '`\$(sysconfdir)`'. '`\$(localstatedir)`' should normally be '`/usr/local/var`', but write it as '`\$(prefix)/var`'.

'`libdir`'
The directory for object files and libraries of object code. Do not install executables here, they probably be-long in '`\$(libexecdir)`' instead. The value of `libdir` should normally be '`/usr/local/lib`', but write it as '`\$(exec_prefix)/lib`'.

'`infodir`'
The directory for installing the Info files for this package. By default, it should be '`/usr/local/info`', but it should be written as '`\$(prefix)/info`'.

'`includedir`'
The directory for installing header files to be included by user programs with the C '`#include`' preprocessor directive. This should normally be '`/usr/local/include`', but write it as '`\$(prefix)/include`'.

Most compilers other than GCC do not look for header files in '`/usr/local/include`'. So installing the header files this way is only useful with GCC. Sometimes this is not a problem because some libraries are only really intended to work with GCC. But some libraries are intended to work with other compilers. They should install their header files in two places, one specified by `includedir` and one specified by `oldincludedir`.

'`oldincludedir`'
The directory for installing '`#include`' header files for use with compilers other than GCC. This should normally be '`/usr/include`'.

The Makefile commands should check whether the value of `oldincludedir` is empty. If it is, they should not try to use it; they should cancel the second installation of the header files. A package should not replace an existing header in this directory unless the header came from the same package. Thus, if your Foo package provides a header file '`foo.h`', then it should install the header file in the `oldincludedir` directory if either (1) there is no '`foo.h`' there, or, (2), the '`foo.h`' that exists came from the Foo package. To tell whether '`foo.h`' came from the Foo package, put a magic string in the file—part of a comment—and grep for that string.

Unix-style `man` pages are installed in one of the following:

‘`mandir`’

The directory for installing the man pages (if any) for this package. It should include the suffix for the proper section of the documentation—usually ‘`1`’ for a utility. It will nor-mally be ‘`/usr/local/man/man1`’ but you should write it as ‘`\$(prefix)/man/man1`’.

‘`man1dir`’

The directory for installing section 1 `man` pages.

‘`man2dir`’

The directory for installing section 2 `man` pages.

‘`...`’

Use these names instead of ‘`mandir`’ if the package needs to install `man` pages in more than one section of the documentation.

WARNING: Don’t make the primary documentation for any GNU software be a `man` page.

Write a manual in Texinfo instead. `man` pages are just for the sake of people running GNU software on Unix which is 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.

And finally, you should set the following variable:

‘`srcdir`’

The directory for the sources being compiled. The value of this variable is normally inserted by the `configure` shell script.

Use the following for example.

```
# Common prefix for installation directories.
# NOTE: This directory must exist when you start the install.
prefix = /usr/local
exec_prefix = $(prefix)
# Where to put the executable for the command ‘gcc’.
bindir = $(exec_prefix)/bin
# Where to put the directories used by the compiler.
```

```
libexecdir = $(exec_prefix)/libexec
# Where to put the Info files.
infodir = $(prefix)/info
```

If your program installs a large number of files into one of the standard user-specified directories, it might be useful to group them into a subdirectory particular to that program. If you do this, you should write the `install` rule to create these subdirectories.

Do not expect the user to include the subdirectory name in the value of any of the variables 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.

16

Quick reference to `make`

The following summary describes the directives, text manipulation functions, and special variables that GNU **`make`** understands and recognizes. See “Special built-in target names” on page 340, “Catalogue of implicit rules” on page 429, and “Summary of options for make” on page 419 for other discusssions.

`define` *`variable`*
`endef`

Define a multi-line, recursively-expanded variable. See “Defining canned command sequences” on page 365.

`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 387.

16: Quick reference to make

`include` *`file`*
: Include another makefile. See “Including other makefiles” on page 318.

`override` *`variable`*`=` *`value`*
`override` *`variable`*`:=` *`value`*
`override` *`variable`*`+=` *`value`*
`override define` *`variable`*
`endef`
: Define a variable, overriding any previous definition, even one from the command line. See “The override directive” on page 383.

`export`
: Tell make to export all variables to child processes by default. See “Communicating variables to a sub-make” on page 361.

`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” on page 361.

`vpath` *`pattern path`*
: Specify a search path for files matching a ‘`%`’ pattern. See “The vpath directive” on page 332.

`vpath` *`pattern`*
: Remove all search paths previously specified for *`pattern`*.

`vpath`
: Remove all search paths previously specified in any `vpath` directive.

The following is a summary of the text manipulation functions (see “Functions for transforming text” on page 395):

`$(subst` *`from`*`,` *`to`*`,` *`text`*`)`
: Replace *`from`* with *`to`* in *`text`*. See “Functions for string substitution and analysis” on page 397.

`$(patsubst` *`pattern`*`,` *`replacement`*`,` *`text`*`)`
: Replace words matching *`pattern`* with *`replacement`* in *`text`*. See “Functions for string substitution and analysis” on page 397.

`$(strip` *`string`*`)`
: Remove excess whitespace characters from *`string`*. See “Functions for string substitution and analysis” on page 397.

`$(findstring find, text)`
: Locate *find* in *text*. See "Functions for string substitution and analysis" on page 397.

`$(filter pattern...,text)`
: Select words in *text* that match one of the *pattern* words. See "Functions for string substitution and analysis" on page 397.

`$(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 397.

`$(sort list)`
: Sort the words in *list* lexicographically, removing duplicates. See "Functions for string substitution and analysis" on page 397.

`$(dir names...)`
: Extract the directory part of each file name. See "Functions for file names" on page 400.

`$(notdir names...)`
: Extract the non-directory part of each file name. See "Functions for file names" on page 400.

`$(suffix names...)`
: Extract the suffix (the last '`.`' and following characters) of each file name. See "Functions for file names" on page 400.

`$(basename names...)`
: Extract the base name (name without suffix) of each file name. See "Functions for file names" on page 400.

`$(addsuffix suffix, names...)`
: Append *suffix* to each word in *names*. See "Functions for file names" on page 400.

`$(addprefix prefix, names...)`
: Prepend *prefix* to each word in *names*. See "Functions for file names" on page 400.

`$(join list1, list2)`
: Join two parallel lists of words. See "Functions for file names" on page 400.

`$(word n, text)`
: Extract the *n*th word (one-origin) of *text*. See "Functions for file names" on page 400.

`$(words text)`
: Count the number of words in *text*. See "Functions for file names" on page 400.

`$(firstword names ...)`
: Extract the first word of *names*. See “Functions for file names” on page 400.

`$(wildcard pattern ...)`
: Find file names matching a shell file name, *pattern* (not a ‘%’ pattern). See “The wildcard function” on page 331.

`$(shell command)`
: Execute a shell command and return its output. See “The shell function” on page 407.

`$(origin variable)`
: Return a string describing how the make variable, *variable*, was defined. See “The origin function” on page 405.

`$(foreach var, words, text)`
: Evaluate *text* with *var* bound to each word in *words*, and concatenate the results. See “The foreach function” on page 403.

The following is a summary of the automatic variables. See “Automatic variables” on page 440 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 451).

`$^`
`$+`
: 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 451). 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 442).

`$(@D)`
`$(@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 `$?`.

The following variables are used specially by GNU **`make`**.

`MAKEFILES`
Makefiles to be read on every invocation of **`make`**. See “The variable, MAKEFILES” on page 320.

`VPATH`
Directory search path for files not found in the current directory.

See “VPATH: search path for all dependencies” on page 332.

`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 354.

`MAKE`
The name with which **`make`** was invoked. Using this variable in commands has special meaning. See “How the MAKE variable works” on page 360.

`MAKELEVEL`
The number of levels of recursion (sub-**`make`**s). See “Communicating variables to a sub-make” on page 361.

`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” on page 361.

`SUFFIXES`
The default list of suffixes before **`make`** reads any makefiles.

17

Complex makefile example

The following is the makefile for the GNU **tar** program. This is a moderately complex makefile.

Because it is the first target, the default goal is ‘all’. An interesting feature of this makefile is that ‘testpad.h’ is a source file automatically created by the testpad program, itself compiled from ‘testpad.c’.

If you type ‘make’ or ‘make all’, then **make** creates the ‘**tar**’ executable, the ‘**rmt**’ daemon that provides remote tape access, and the ‘**tar.info**’ Info file.

If you type ‘make install’, then **make** not only creates ‘**tar**’, ‘**rmt**’, and ‘**tar.info**’, but also installs them.

If you type ‘make clean’, then **make** removes the ‘**.o**’ files, and the ‘**tar**’, ‘**rmt**’, ‘**testpad**’, ‘**testpad.h**’, and ‘**core**’ files.

If you type ‘make distclean’, then **make** not only removes the same files as does ‘make clean’ but also the ‘**TAGS**’, ‘**Makefile**’, and ‘**config.status**’ files. (Although it is not evident, this makefile (and ‘**config.status**’) is generated by the user with the configure program which is provided in the **tar** distribution; 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 ioctls).
# -DNO_REMOTE            If you do not have a remote shell
#                         or rexec.
# -DUSE_REXEC            To use rexec for remote tape
#                         operations instead of
#                         forking rsh or remsh.
# -DVPRINTF_MISSING      If you lack vprintf function
#                         (but have _doprnt).
# -DDOPRNT_MISSING       If you lack _doprnt function.
#                         Also need to define
#                         -DVPRINTF_MISSING.
# -DFTIME_MISSING        If you lack ftime system call.
# -DSTRSTR_MISSING       If you lack strstr function.
# -DVALLOC_MISSING       If you lack valloc function.
# -DMKDIR_MISSING        If you lack mkdir and
#                         rmdir system calls.
# -DRENAME_MISSING       If you lack rename system call.
# -DFTRUNCATE_MISSING    If you lack ftruncate
#                         system call.
# -DV7                    On Version 7 Unix (not
#                         tested in a long time).
# -DEMUL_OPEN3           If you lack a 3-argument version
#                         of open, and want to emulate it
#                         with system calls you do have.
# -DNO_OPEN3             If you lack the 3-argument open
#                         and want to disable the tar -k
#                         option instead of emulating open.
# -DXENIX                If you have sys/inode.h
#                         and need it 94 to be included.
DEFS = -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
       -DVPRINTF_MISSING -DBSD42
# Set this to rtapelib.o unless you defined NO_REMOTE,
# in which case make it empty.
```

17: Complex makefile example

```
RTAPELIB = rtapelib.o
LIBS =
DEF_AR_FILE = /dev/rmt8
DEFBLOCKING = 20

CDEBUG = -g
CFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \
         -DDEF_AR_FILE=\ “$(DEF_AR_FILE)\” \
         -DDEFBLOCKING=$(DEFBLOCKING)
LDFLAGS = -g

prefix = /usr/local
# Prefix for each installed program,
# normally empty or ‘g’.
binprefix =

# The directory to install tar in.
bindir = $(prefix)/bin

# The directory to install the info files in.
infodir = $(prefix)/info

#### End of system configuration section. ####

SRC1 = tar.c create.c extract.c buffer.c \
       getoldopt.c update.c gnu.c mangle.c
SRC2 = version.c list.c names.c diffarch.c \
       port.c wildmat.c getopt.c
SRC3 = getopt1.c regex.c getdate.y
SRCS = $(SRC1) $(SRC2) $(SRC3)
OBJ1 = tar.o create.o extract.o buffer.o \
       getoldopt.o update.o gnu.o mangle.o
OBJ2 = version.o list.o names.o diffarch.o \
       port.o wildmat.o getopt.o
OBJ3 = getopt1.o regex.o getdate.o $(RTAPELIB)
OBJS = $(OBJ1) $(OBJ2) $(OBJ3)
AUX  = README COPYING ChangeLog Makefile.in \
       makefile.pc configure configure.in \
       tar.texinfo tar.info* texinfo.tex \
       tar.h port.h open3.h getopt.h regex.h \
```

```
        rmt.h rmt.c rtapelib.c alloca.c \
        msd_dir.h msd_dir.c tcexparg.c \
        level-0 level-1 backup-specs testpad.c

all:    tar rmt tar.info

tar:    $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
rmt:     rmt.c
         $(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.c
tar.info: tar.texinfo
         makeinfo tar.texinfo

install: all
         $(INSTALL) tar $(bindir)/$(binprefix)tar
         -test ! -f rmt || $(INSTALL) rmt /etc/rmt
         $(INSTALLDATA) $(srcdir)/tar.info* $(infodir)

$(OBJS): tar.h port.h testpad.h
regex.o buffer.o tar.o: regex.h
# getdate.y has 8 shift/reduce conflicts.

testpad.h: testpad
         ./testpad

testpad: testpad.o
         $(CC) -o $@ testpad.o

TAGS:    $(SRCS)
         etags $(SRCS)

clean:
         rm -f *.o tar rmt testpad testpad.h core

distclean: clean
         rm -f TAGS Makefile config.status
realclean: distclean
         rm -f tar.info*
shar: $(SRCS) $(AUX)
         shar $(SRCS) $(AUX) | compress \
```

```
        > tar-‘sed -e ’/version_string/!d’ \
                        -e ’s/[ˆ0-9.]*\([0-9.]*\).*/\1/’ \
                        -e q
                        version.c‘.shar.Z
dist: $(SRCS) $(AUX)
        echo tar-‘sed \
                -e ’/version_string/!d’ \
                -e ’s/[ˆ0-9.]*\([0-9.]*\).*/\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 $$X ; \
                sed ’s/$$/ˆM/’ $$X \
                > tmp.dir/$$X ; done
        cd tmp.dir ; zoo aM ../tar.zoo *
        -rm -rf tmp.dirIndex
```

Index

Symbols

Numerics

A

B

D

E

Index

F

I

J

K

M

N

Index

O

S

Index

W

X