GNU Make

A Program for Directing Recompilation

February 1988

Richard M. Stallman

1. Overview of `make`

The purpose of the `make` utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. This manual describes the GNU implementation of `make`.

Our examples show C programs, since they are most common, but you can use `make` with any programming language whose compiler can be run with a shell command. In fact, `make` is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use `make`, you must write a file called the *makefile* that describes the relationships among files in your program, and the states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The `make` program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

Command arguments to `make` can be used to control which files should be recompiled, or how. See chapter 8 [Running], page 47.

GNU `make` was implemented by Richard Stallman and Roland McGrath.

1.1 Simple Example of `make`

Suppose we have a text editor consisting of eight C source files and three header files. We need a makefile to tell `make` how to compile and link the editor. Assume that all the C files include '`defs.h`', but only those defining editing commands include '`commands.h`' and only low level files that change the editor buffer include '`buffer.h`'.

To recompile the editor, each changed C source file must be recompiled. If a header file has changed, in order to be safe each C source file that includes the header file must be recompiled. 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.

Here is a straightforward makefile that describes these criteria and says how to compile and link when the time comes:

```
edit : main.o kbd.o commands.o display.o \
              insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o commands.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
commands.o : command.c defs.h command.h
        cc -c commands.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
```

We split each long line into two lines using a backslash-newline; this is like using one long line, but is easier to read.

Each file that is generated by a program—that is to say, each file except for source files—is the *target* of a *rule* (see chapter 3 [Rules], page 11). (In this example, these are the object files such as '`main.o`', '`kbd.o`', etc., and the executable file '`edit`'.) The target appears at the beginning of a line, followed by a colon.

After the colon come the target's *dependencies*: all the files that are used as input when the target file is updated. A target file needs to be recompiled or relinked if any of its dependencies changes. In addition, any dependencies that are themselves automatically generated should be updated first. In this example, '`edit`' depends on each of the eight object files; the object file 'main.o' depends on the source file 'main.c' and on the header file 'defs.h'.

By default, make starts with the first rule (not counting rules whose target names start with '.'). This is called the *default goal*. Therefore, we put the rule for the executable program 'edit' first. The other rules are processed because their targets appear as dependencies in the goal.

After each line containing a target and dependencies come one or more lines of shell commands that say how to update the target file. These lines start with a tab to tell make that they are command lines. But 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 you have specified when the target file needs to be updated.

1.1.1 How make Processes This Makefile

After reading the makefile, make begins its real work by processing the first rule, the one for relinking 'edit'; but before it can fully process this rule, it must process the rules for the files 'edit' depends on: all the object files. Each of these files is processed according to its own rule. These rules say to update the '.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.

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 nothing needs to be done. But automatically generated C programs, such as made by Yacc, would be updated by their own rules at this time.

After recompiling whichever object files need it, make can now decide 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' will be relinked.

1.1.2 Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for 'edit' (repeated here):

```
edit : main.o kbd.o commands.o display.o \
              insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o commands.o display.o \
```

```
                insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a *variable*. Variables allow a text string to be defined once and substituted in multiple places later (see chapter 5 [Variables], page 31).

It's standard practice for every makefile to have a variable named `objects`, `OBJECTS` or `OBJ` which is a list of all object file names. We would define such a variable `objects` with a line like this in the makefile:

```
objects = main.o kbd.o commands.o display.o \
              insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable's value by writing '`$(objects)`' (see chapter 5 [Variables], page 31). Here is how the rule for `edit` looks as a result:

```
edit : $(objects)
        cc -o edit $(objects)
```

1.1.3 Letting `make` Deduce the Commands

It is not necessary to spell out the commands for compiling the individual C source files, because `make` can figure them out: it has an *implicit rule* for updating a '`.o`' file from a correspondingly named '`.c`' file using a '`cc -c`' command. For example, it will use the command '`cc -c main.c -o main.o`' to compile '`main.c`' into '`main.o`'. We can therefore omit the commands from the rules for the object files. See chapter 9 [Implicit], page 55.

When a '`.c`' file is used automatically in this way, it is also automatically added to the list of dependencies. We can therefore omit the '`.c`' files from the dependencies, provided we omit the commands.

Here is the entire example, with both of these changes, and a variable `objects` as suggested above:

```
objects =  main.o kbd.o commands.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
commands.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
```

This is how we would write the makefile in actual practice.

1.1.4 Another Style of Makefile

Since the rules for the object files specify only dependencies, no commands, one can alternatively combine them by dependency instead of by target. Here is what it looks like:

```
objects =  main.o kbd.o commands.o display.o \
 insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects): defs.h
kbd.o commands.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here '`defs.h`' is given as a dependency of all the object files; '`commands.h`' and '`buffer.h`' are dependencies of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

2. Writing Makefiles

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

2.1 What Makefiles Contain

Makefiles contain four kinds of things: *rules*, *variable definitions*, *directives* and *comments*. Rules, variables and directives are described at length in later chapters.

- A rule says when and how to remake one or more files, called the rule's *targets*. It lists the other files that the targets *depend on*, and may also give commands to use to create or update the targets. See chapter 3 [Rules], page 11.
- 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 (see section 1.1 [Simple], page 1) shows a variable definition for `objects` as a list of all object files. See chapter 5 [Variables], page 31, for full details.
- A directive is a command for `make` to do something special while reading the makefile. These include:
  - Reading another makefile (see section 2.3 [Include], page 8).
  - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see chapter 6 [Conditionals], page 37).
  - Defining a variable from a verbatim string containing multiple lines (see section 5.5 [Defining], page 34).
- '`#`' in a line of a makefile starts a comment. It and the rest of the line are ignored. 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.

2.2 What Name to Give Your Makefile

By default, when `make` looks for the makefile, it tries the names '`./makefile`' or '`./Makefile`' in that order. So normally you call your makefile by one of these two names, and `make` finds it automatically. We recommend '`Makefile`' because it appears prominently near the beginning of a directory listing.

If `make` finds neither of these two 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.

If you want to use a nonstandard name for your makefile, you can specify the makefile name with the '`-f`' option. The arguments '`-f` *name*' tell `make` to read the file *name* as the makefile. If you use more than one '`-f`' option, you can specify several makefiles. All the makefiles are effectively concatenated in the order specified. The default makefile names '`./makefile`' and '`./Makefile`' are not used if you specify '`-f`'.

If the environment variable `MAKEFILES` is defined, `make` considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive: various directories are searched for those files and the default goal is never taken from them. See section 2.3 [Include], page 8. In addition, it is not an error if the files listed in `MAKEFILES` are not found.

The main use of `MAKEFILES` is in communication between recursive invocations of `make` (see section 4.5 [Recursion], page 26). It usually isn't 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.

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.

2.3 Including Other Makefiles

The `include` directive tells `make` to suspend reading the current makefile and read another makefile before continuing. The directive is a line in the makefile that looks like this:

```
include filename
```

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 *filename*; extra whitespace is ignored there and at the end of the directive. A comment starting with '`#`' is allowed at the end of the line.

Reading of the containing makefile is temporarily suspended while the file *filename* is read as a makefile. When that is finished, `make` goes on with reading the makefile in which the directive appears.

The default goal target is never taken from an included makefile (see section 8.2 [Goals], page 47).

One occasion for using `include` directives is when several programs, handled by individual makefiles in various directories, need to use a common set of variable definitions (see section 5.3 [Setting], page 32) or pattern rules (see section 9.5 [Pattern Rules], page 61).

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`' option are searched (see section 8.7 [Options], page 51). Then the following directories (if they exist) are searched, in this order: '`/usr/gnu/include`', '`/usr/local/include`', '`/usr/include`'.

3. Writing Rules

A *rule* appears in the makefile and says when and how to remake certain files, called the rule's *targets* (usually only one per rule). It lists the other files that are the *dependencies* of the target, and *commands* to use to create or update the target.

The order of rules is not significant, except for determining the *default goal*: the target for **make** to consider, if you do not otherwise specify one. The default goal comes from the first rule (not counting included makefiles) whose target does not start with a period. Therefore, the first rule is normally one for compiling the entire program or all the programs described by the makefile. See section 8.2 [Goals], page 47.

3.1 Rule Syntax

In general, a rule looks like this:

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

or like this:

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

The *targets* are file names, separated by spaces. Wild card characters may be used (see section 3.2 [Wildcards], page 12) and a name of the form '`a(m)`' represents member *m* in archive file *a* (see section 10.1 [Archive Members], page 69). Usually there is only one target per rule, but occasionally there is a reason to have more (see section 3.7 [Multiple Targets], page 19).

The *command* lines start with a tab character. The first command may appear on the line after the dependencies, with a tab character, or may appear on the same line, with a semicolon. Either way, the effect is the same. See chapter 4 [Commands], page 23.

Because dollar signs are used to start variable references, if you really want a dollar sign in the rule you must write two of them (`$$`). See chapter 5 [Variables], page 31. A long line may be split by inserting a backslash followed by a newline, but this is not required, as there is no limit on the length of a line.

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 are allowed here too.) A target is out of date if it does not exist or if it is older than any of the dependencies (by comparison of last-modification times). The idea is that the contents of the target file are computed based on information in the dependencies, so if any of the dependencies changes the contents of the existing target file are no longer necessarily valid.

How to remake is specified by *commands*. These are lines to be executed by the shell (normally `sh`), but with some extra features (see chapter 4 [Commands], page 23).

3.2 Using Wildcards 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`.

Wildcard expansion happens automatically in targets, in dependencies, and in commands. 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`.

3.2.1 Wildcard Examples

Wildcards can be used in the commands of a rule. For example, here is a rule to delete all the object files:

```
clean:
```

```
        rm -f *.o
```

Wildcards are also useful in the dependencies of a rule. With the following rule in the makefile, '`make print`' will print all the '`.c`' files that have changed since the last time you printed them:

```
print: *.c
        lpr -p $?
        touch print
```

This rule uses '`print`' as an empty target file; see section 3.5 [Empty Targets], page 18.

Wildcard expansion does not happen when you define a variable. Thus, if you write this:

```
objects=*.o
```

then the value of the variable `objects` is the actual string '`*.o`'. However, if you use the value of `objects` in a target, dependency or command, wildcard expansion will take place at that time.

3.2.2 Pitfalls of Using Wildcards

Now here is an example of a naive way of using wildcard expansion, that does not do what you would intend. Suppose you would like to say that the executable file '`foo`' is made from all the object files in the directory, and you write this:

```
objects=*.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

The value of `objects` is the actual string '`*.o`'. Wildcard expansion happens in the rule for '`foo`', so that each *existing* '`.o`' file becomes a dependency of '`foo`' and will be recompiled if necessary.

But what if you delete all the '`.o`' files? Then '`*.o`' will expand into *nothing*. The target '`foo`' will have no dependencies and would be remade by linking no object files. This is not what you want!

Actually you can use wildcard expansion for this purpose, but you need more sophisticated techniques, including the `wildcard` function and string substitution. These are described in the following section.

3.2.3 The Function `wildcard`

Wildcard expansion happens automatically in rules. But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function. If you want to do wildcard expansion in such places, you need to use the `wildcard` function, like this:

```
$(wildcard pattern)
```

This string, used anywhere in a makefile, is replaced by a space-separated list of names of existing files that match the pattern *pattern*.

One use of the `wildcard` function is to get a list of all the C source files in a directory, like this:

```
$(wildcard *.c)
```

We can change the list of C source files into a list of object files by substituting '`.o`' for '`.c`' in the result, like this:

```
$(subst .c,.o,$(wildcard *.c))
```

Here we have used another function, `subst` (see section 7.2 [Text Functions], page 42).

Thus, a makefile to compile all C source files in the directory and then link them together could be written as follows:

```
objects=$(subst .c,.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(LDFLAGS) $(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.)

3.3 Searching Directories for Dependencies

For large systems, it is often desirable to put sources in a separate directory from the binaries. The VPATH feature makes this easier.

The value of the variable VPATH is a list of directories which make should search (in the order specified) for dependency files. The directory names are separated by colons. For example:

```
VPATH = src:../headers
```

specifies a path containing two directories, 'src' and '../headers'.

When a file listed as a dependency does not exist in the current directory, the directories listed in VPATH are searched 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 as if they all existed in the current directory.

Using the value of VPATH set in the previous example, a rule like this:

```
foo.o : foo.c
```

is interpreted as if it were written like this:

```
foo.o : src/foo.c
```

assuming the file 'foo.c' does not exist in the current directory but is found in the directory 'src'.

But what about the rule's commands? The VPATH directory search cannot change the commands; they will execute as written. You need to write the commands so that they will use the file names that make finds. This is done with the *automatic variables* such as '$^' (see section 9.5.2 [Automatic], page 62). For instance, the value of '$^' is a list of all the dependencies of the rule, including the names of the directories in which they were found, and the value of '$@' is the target. Thus:

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

The variable CFLAGS exists so you can specify flags for C compilation by changing its value; we use it here for consistency so it will affect all C compilations uniformly. (see section 9.3 [Implicit

Variables], page 58).

Often the dependencies include header files as well, which you don't want to mention in the commands. The function `firstword` can be used to extract just the first dependency from the entire list, as shown here (see section 7.4 [Filename Functions], page 43):

```
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $(firstword $^) -o $@
```

Here the value of '`$^`' is something like '`src/foo.c ../headers/defs.h hack.h`', from which '`$(firstword $^)`' extracts just '`src/foo.c`'.

3.3.1 Directory Search and Implicit Rules

The search through the directories in `VPATH` happens also during consideration of implicit rules (see chapter 9 [Implicit], page 55).

For example, when a file '`foo.o`' has no explicit rule, `make` considers implicit rules, such as to compile '`foo.c`' if that file exists. If such a file is lacking in the current directory, the directories in `VPATH` 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 applicable.

The commands of all the built-in 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.

3.4 Phony Targets

A phony target is one that is not really the name of a file. It is only a name for some commands to be executed when explicitly requested.

If you write a rule whose commands will not create the target file, the commands will be executed every time the target comes up for remaking. Here is an example:

```
clean:
        rm *.o temp
```

Because the `rm` command does not create a file named '`clean`', probably no such file will ever exist. Therefore, the `rm` command will be executed every time you say '`make clean`'.

The phony target will cease to work if anything ever does create a file named '`clean`' in this directory. Since there are no dependencies, the '`clean`' would be considered up-to-date and its commands would not be executed. To avoid this problem, you can explicitly declare the target to be phony, using the special target `.PHONY` (see section 3.6 [Special Targets], page 18) as follows:

```
.PHONY : clean
```

Once this is done, `make` will run the commands regardless of whether there is a file named '`clean`'.

A phony target should not be a dependency of a real target file; strange things can result from that. As long as you don't do that, the phony target commands will be executed only when the phony target is a goal (see section 8.2 [Goals], page 47).

Phony targets can have dependencies. When one directory contains multiple programs, it is most convenient to describe all of the programs in one makefile '`./Makefile`'. Since the target remade by default will be the first one in the makefile, it is common to make this a phony target named '`all`' and give it, as dependencies, all the individual programs. For example:

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

Now you can say `make` to remake all three programs, or specify as arguments the ones to remake (as in '`make prog1 prog3`').

When one phony target is a dependency of another, it serves as a subroutine of the other. For example, here '`make cleanall`' will delete the object files, the difference files, and the file '`program`':

```
cleanall : cleanobj cleandiff
        rm program
```

```
cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

3.5 Empty Target Files to Record Events

The *empty target* is a variant of the phony target; it is used to hold commands for an action that you request explicitly from time to time. Unlike a phony target, this target file can really exist; but the file's contents do not matter, and usually are empty.

The purpose of the empty target file is to record, with its last-modification-time, when the rule's commands were last executed. It does so because one of the commands is a `touch` command to update the target file.

The empty target file must have some dependencies. When you ask to remake the empty target, the commands are executed if any dependency is more recent than the target; in other words, if a dependency has changed since the last time you remade the target. Here is an example:

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

With this rule, '`make print`' will execute the `lpr` command if either source file has changed since the last '`make print`'. The automatic variable '`$?`' is used to print only those files that have changed (see section 9.5.2 [Automatic], page 62).

3.6 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 date is. See section 3.4 [Phony Targets], page 16.

`.SUFFIXES`
: The dependencies of the special target `.SUFFIXES` are the list of suffixes to be used in checking for suffix rules (see section 9.7 [Suffix Rules], page 66).

`.DEFAULT`
: The commands specified for `.DEFAULT` are used for any target for which no other commands are known (either explicitly or through an implicit rule). If `.DEFAULT` commands are specified, every nonexistent file mentioned as a dependency will have these commands executed on its behalf. See section 9.8 [Search Algorithm], page 67.

`.PRECIOUS`
: The targets which `.PRECIOUS` depends on are given this special treatment: if **make** is killed or interrupted during the execution of their commands, the target is not deleted. See section 4.4 [Interrupts], page 25.

`.IGNORE`
: Simply by being mentioned as a target, `.IGNORE` says to ignore errors in execution of commands. The dependencies and commands for `.IGNORE` are not meaningful.

  '`.IGNORE`' exists for historical compatibility. Since `.IGNORE` affects every command in the makefile, it is not very useful; we recommend you use the more selective ways to ignore errors in specific commands (see section 4.3 [Errors], page 24).

`.SILENT`
: Simply by being mentioned as a target, `.SILENT` says not to print commands before executing them. The dependencies and commands for `.SILENT` are not meaningful.

  '`.SILENT`' exists for historical compatibility. We recommend you use the more selective ways to silence specific commands (see section 4.1 [Echoing], page 23).

An entire class of special targets have names made of the concatenation of two implicit rule suffixes (two members of the list of dependencies of `.SUFFIXES`). Such special targets are suffix rules, an obsolete way of defining implicit rules (but a way still widely used). In principle, any target name could be special in this way if you break it in two and add both pieces to the suffix list. In practice, suffixes normally begin with '`.`', so these special target names also begin with '`.`'. See section 9.7 [Suffix Rules], page 66.

3.7 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. This is useful in two cases.

- You want just dependencies, no commands. For example:

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

  gives an additional dependency to each of the three object files mentioned.

- Identical commands work for all the targets. The automatic variable '`$@`' can be used to substitute the target to be remade into the commands (see section 9.5.2 [Automatic], page 62). For example:

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

  is equivalent to

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

  Here we assume the hypothetical program `generate` makes two types of output, one if given '`-big`' and one if given '`-little`'.

3.8 Multiple Rules for One Target

One file can be the target of several rules if at most one rule has commands. The other rules can only have dependencies. 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.

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

```
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 say 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 could set with a command argument to `make` (see section 8.5 [Overriding], page 50). For example,

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

means that the command '`make extradeps=foo.h`' will consider '`foo.h`' as a dependency of each object file, but plain '`make`' will not.

If none of the explicit rules for a target has commands, then `make` searches for an applicable implicit rule to find some commands. See chapter 9 [Implicit], page 55.

3.9 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.

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.

Each double-colon rule should specify commands; if it does not, an implicit rule will be used if one applies. See chapter 9 [Implicit], page 55.

4. Writing the Commands in Rules

The commands of a rule consist of shell command lines to be executed one by one. Each command line must start with a tab, except that the first command line may be attached to the target-and-dependencies line with a semicolon in between. Blank lines and lines of just comments may appear among the command lines; they are ignored.

Users use many different shell programs, but commands in makefiles are always interpreted by '`/bin/sh`' unless the makefile specifies otherwise.

Whether comments can be written on command lines, and what syntax they use, is under the control of the shell that is in use. If it is '`/bin/sh`', a '`#`' at the start of a word starts a comment.

4.1 Command Echoing

Normally `make` prints each command line before it is executed. We call this *echoing* because it gives the appearance that you are typing the commands yourself.

When a line starts with '`@`', it is normally not echoed. 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.

When `make` is given the flag '`-n`', echoing is all that happens, no execution. See section 8.7 [Options], page 51. 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`' flag to `make` prevents all echoing, as if all commands started with '`@`'. A rule in the makefile for the special target `.SILENT` has the same effect (see section 3.6 [Special Targets], page 18). `.SILENT` is essentially obsolete since '`@`' is more general.

4.2 Command Execution

When it is time to execute commands to update a target, they are executed one at a time by making a new subshell for each line. (In practice, `make` may take shortcuts that do not affect the results.)

This implies that shell commands such as `cd` that set variables local to each process will not affect the following command lines. If you want to use `cd` to affect the next command, put the two on a single line with a semicolon between them. Then `make` will consider them a single command and pass them, together, to a shell which will execute them in sequence. For example:

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

If you would like to split a single shell command into multiple lines of text, you must use a backslash at the end of all but the last subline. Such a sequence of lines is combined into a single line, by deleting the backslash-newline sequences, before passing it to the shell. Thus, the following is equivalent to the preceding example:

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

The program used as the shell is taken from the variable `SHELL`. By default, the program '`/bin/sh`' is used.

Unlike most variables, the variable `SHELL` will not be set from the environment, except in a recursive `make`. This is because the environment variable `SHELL` is used to specify your personal choice of shell program for interactive use. It would be very bad for personal choices like this to affect the functioning of makefiles. See section 5.6 [Environment], page 35.

The value of the variable `SHFLAGS` is used as additional command arguments to give to the shell each time it is run. By default, the value is empty. This variable also is not set from the environment except in recursive use of `make`.

4.3 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, or after the last command line 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 it does not matter whether a command fails. For example, you may use the **mkdir** command to insure 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.

When **make** is run with the '-i' flag, errors are ignored in all commands of all rules. A rule in the makefile for the special target .IGNORE has the same effect. These ways of ignoring errors are obsolete because '-' is more general.

When errors are to be ignored, because of either a '-' or the '-i' flag, **make** treats an error return just like success.

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' flag is specified, **make** continues to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, '**make -k**' will continue compiling other object files even though it already knows that linking them will be impossible. See section 8.7 [Options], page 51.

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. '-k' 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's **compile** command passes the '-k' flag by default.

4.4 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 date 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. Otherwise, a partially written file could appear to be valid, since it is more recent than the dependencies.

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 date/time (its contents do not matter) or will cause trouble if it ever fails to exist.

4.5 Recursive Use of make

Recursive use of `make` means using `make` as a command in a makefile. This technique is useful when you want separate makefiles for various subsystems that compose a larger system. For example, suppose you have a subdirectory '`subdir`' which has its own makefile, and you would like the containing directory's makefile to run `make` on the subdirectory.

You can do it by writing this:

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

or, equivalently, this (see section 8.7 [Options], page 51):

```
subsystem:
        $(MAKE) -c subdir
```

That's all you have to write in the makefile to cause the sub-`make` to be run if you do `make subsystem`, but there are other things you should know about how this does its job and how the sub-`make` relates to the top-level `make`.

The commands above use the variable `MAKE`, whose value 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. Also, any arguments that define variable values are added to `MAKE`, so the sub-`make` gets them too. Thus, if you do '`make CFLAGS=-O`', so that all C-compilations will be optimized, the sub-`make` is run with '`cd subdir; /bin/make CFLAGS=-O`'.

The flag options you give to the top-level `make` are passed down to the sub-`make` automatically, through the variable `MAKEFLAGS` as described below.

All the other variable values of the top-level `make` are passed to the sub-`make` through the environment. These variables are defined in the sub-`make` as defaults, but do not override what is specified in the sub-`make`'s makefile. The way this works is that `make` adds each variable and its value to the environment for running each command. (Variables whose names start with non-alphanumeric characters are left out.) The sub-`make`, in turn, uses the environment to initialize its table of variable values. See section 5.6 [Environment], page 35.

As a special feature, the variable `MAKELEVEL` is changed when it is passed down from level to level. This variable's value is a string which is the depth of the level as a decimal number. The value is '`0`' for the top-level `make`; '`1`' for a sub-`make`, '`2`' for a sub-sub-`make`, and so on. The incrementation happens when `make` sets up the environment for a command.

The main use of `MAKELEVEL` is to test it in a conditional directive (see chapter 6 [Conditionals], page 37); this way you can write a makefile that behaves one way if run recursively and another way if run directly by you.

Flags such as '`-s`' and '`-k`' are passed automatically to the sub-`make` through the variable `MAKEFLAGS`. This variable is set up automatically by `make` to contain the flag letters that `make` received. Thus, if you do '`make -ks`' then `MAKEFLAGS` gets the value '`ks`'.

As a consequence, every sub-`make` gets a value for `MAKEFLAGS` in its environment. In response, it takes the flags from that value and processes them as if they had been given as arguments. See section 8.7 [Options], page 51.

The options '`-c`', '`-d`', '`-f`', '`-I`', '`-o`', and '`-p`' are not put into `MAKEFLAGS`; these options are not passed down.

If you don't want to pass the other the flags down, you must change the value of `MAKEFLAGS`, like this:

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

A similar variable `MFLAGS` exists also, for historical compatibility. It has the same value as `MAKEFLAGS` except that a hyphen is added at the beginning if it is not empty. `MFLAGS` was traditionally used explicitly in the recursive `make` command, like this:

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

but now `MAKEFLAGS` makes this usage redundant.

What about '`make -t`'? (See section 8.4 [Instead of Execution], page 49.) Following the usual definition of '`-t`', this would create a file named '`subsystem`'. What you really want it to do is run '`cd subdir; make -t`'; but that would require executing the command, and '`-t`' says not to execute commands.

The paradox is resolved by a special `make` feature: whenever a command uses the variable `MAKE`, the flags '`-t`', '`-n`' or '`-q`' do not apply to that rule. The commands of that rule are executed normally despite the presence of a flag that causes most commands not to be run. These flags are passed along via `MAKEFLAGS`, so your request to touch the files, or print the commands, is propagated to the subsystem.

If the environment variable `MAKEFILES` is defined, `make` considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive: various directories are searched for those files and the default goal is never taken from them. See section 2.3 [Include], page 8. In addition, it is not an error if the files listed in `MAKEFILES` are not found.

The main use of `MAKEFILES` is with recursive invocation of `make`. The outer `make` can set `MAKEFILES` to influence recursive `make` levels.

If you are running `make` over a large directory tree, the `-w` option can make understanding the output a lot easier by showing each directory as it is entered and exited. For example, if `make -w` is run in the directory '`/u/gnu/make`', `make` will print a line of the form

```
#### Entering /u/gnu/make ####
```

before doing anything else, and a line of the form

```
#### Exiting /u/gnu/make ####
```

after finishing with the makefile.

4.6 Defining Canned Command Sequences

When the same sequence of commands is useful in making various targets, you can define it as a canned sequence with the **define** directive, and refer to the canned sequence from the rules for those targets. The canned sequence is actually a variable, so the name must not conflict with other variable names.

Here is an example of defining a canned sequence of commands:

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

Here `run-yacc` is the name of the variable being defined; `endef` marks the end of the definition; the lines in between are the commands. The `define` directive does not expand variable references and function calls in the canned sequence; the '`$`' characters, parentheses, variable names, and so on, all become part of the value of the variable you are defining. See section 5.5 [Defining], page 34, for a complete explanation of `define`.

The first command in this example runs Yacc on the first dependency (of whichever rule uses the canned sequence). The output file from Yacc is always named '`y.tab.c`'. The second command moves the output to the rule's target file name.

To use the canned sequence, substitute the variable into the commands of a rule. You can substitute it like any other variable (see section 5.1 [Reference], page 31). But usually substitution alone is not enough, because the commands in a canned sequence typically contain variable references that should be expanded each time the canned sequence is used. To make this work, you need to use the `expand` function when you substitute the sequence (see section 7.3 [Expand Function], page 43). Here is how it looks:

```
foo.c : foo.y
        $(expand $(run-yacc))
```

The `expand` function will substitute '`foo.y`' for the variable '`$^`' when it occurs in `run-yacc`'s value, and '`foo.c`' for '`$@`'.

This is a realistic example, but this particular one is not needed in practice because `make` has an implicit rule to figure out these commands based on the file names involved. See chapter 9

[Implicit], page 55.

5. How to Use Variables

A *variable* is a name defined within `make` to represent a string of text, called the variable's *value*. These values can be substituted by explicit request into targets, dependencies, commands and other parts of the makefile.

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 characters not containing '`:`', '`#`', '`=`', tab characters or leading or trailing spaces. However, variable names containing nonalphanumeric characters should be avoided, as they may be given special meanings in the future.

It is traditional to use upper case letters in variable names, but we recommend using lower case letters for variable names that serve internal purposes in the makefile, and reserving upper case for parameters that control implicit rules or for parameters that the user should override with command options (see section 8.5 [Overriding], page 50).

5.1 Reference to Variables

To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either '`$(foo)`' or '`${foo}`' is a valid reference to the variable `foo`. This special significance of '`$`' is why you must write '`$$`' to have the effect of a single dollar sign in a file name or command.

Variable references can be used in any context: targets, dependencies, commands, most directives, and new variable values. Here is a common kind of example, where a variable holds the names of all the object files in a program:

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

Variable references work by strict textual substitution. Thus, the rule

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

could be used to compile a C program '`prog.c`'. (Since spaces around the variable value are ignored in variable assignments, the value of `foo` is precisely '`c`'.)

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 with the automatic variables (see section 9.5.2 [Automatic], page 62).

5.2 How Variables Get Their Values

Variables can get values in several different ways:

- You can specify an overriding value when you run `make`. See section 8.5 [Overriding], page 50.
- You can specify a value in the makefile, either with an assignment (see section 5.3 [Setting], page 32) or with a verbatim definition (see section 5.5 [Defining], page 34).
- Values are inherited from the environment. See section 5.6 [Environment], page 35.
- Several *automatic* variables are given new values for each rule. See section 9.5.2 [Automatic], page 62.
- Several variables have constant initial values. See section 9.3 [Implicit Variables], page 58.

5.3 Setting Variables

To set a variable from the makefile, write a line starting with the variable name followed by '`=`'. Whatever follows the '`=`' on the line becomes the value. For example,

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

defines a variable named `objects`. Spaces around the variable name are ignored, and so are spaces after the '`=`' or at the end of the line.

The line that sets the variable can contain variable references. Such references are replaced by their values before the new variable is set. The value given to the new variable does not contain variable references; it contains the substuted values. Thus,

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

is equivalent to

```
x = foo
y = foo bar
```

This gives you a way to introduce leading or trailing spaces into variable values. Such spaces are discarded from your input before substitution of variable references and function calls; this means you can include leading or trailing spaces in a variable value by protecting them with variable references, like this:

```
nullstring=
space=$(nullstring) $(nullstring)
```

Here the value of the variable `space` is precisely one space.

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 can be set by you in the usual ways (see section 9.3 [Implicit Variables], page 58). Several special variables are set automatically to a new value for each rule; these are called the *automatic* variables (see section 9.5.2 [Automatic], page 62).

5.4 The `override` Directive

If a variable has been set with a command argument (see section 8.5 [Overriding], page 50), then ordinary assignments in the makefile are ignored. If you want to set the variable in the makefile even though it was set with a command argument, you can use an `override` directive, which is a line that looks like this:

```
override variable = value
```

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 = $(CFLAGS) -g
```

5.5 Defining Variables Verbatim

Another way to set the value of a variable is to use the **define** directive. This directive has a different syntax and provides different features, and its intended use is for defining canned sequences of commands (see section 4.6 [Sequences], page 29). But the variables made with **define** are just like those made the usual way. Only the variable's value matters.

The **define** directive is followed on the same line the name of the variable and nothing more. The value to give the variable appears on the following lines. These lines are used verbatim; the character '$' is not treated specially and whitespace is not changed. The end of the value is marked by a line containing just the word **endef**.

```
define two-lines
echo foo
echo $100
endef
```

Aside from syntax, there are two differences between **define** and ordinary variable assignments:

- The value assigned in an ordinary variable assignment is scanned for variable references and function calls, which are expanded. The **define** commands are used verbatim, with no replacement.
- The value in an ordinary assignment cannot contain a newline. The text in a **define** can be multiple lines; the newlines that separate the lines become part of the variable's value. (The final newline which is always present does not become part of the variable value.)

5.6 Variables from the Environment

Variables in `make` can come from the environment with which `make` is run. Every environment variable that `make` sees when it starts up is transformed into a `make` variable with the same name and value. But an explicit assignment in the makefile, or with a command argument, overrides the environment. (If the '`-e`' flag is specified, then values from the environment override assignments in the makefile. See section 8.7 [Options], page 51.)

By setting the variable `CFLAGS` in your environment, you can cause all C compilations in most makefiles to use the compiler switches you prefer. This is safe for variables with standard or conventional meanings because you know that no makefile will use them for other things. (But this is not totally reliable; some makefiles set `CFLAGS` explicitly and therefore are not affected by the value in the environment.)

When `make` is invoked recursively, variables defined in the outer invocation are automatically passed to inner invocations through the environment (see section 4.5 [Recursion], page 26). This is the main purpose of turning environment variables into `make` variables, and it requires no attention from you.

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` (and `SHFLAGS`) if the value of `MAKELEVEL` is zero (which is normally true except in recursive invocations of `make`).

6. 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 with another, or the value of a variable with a constant string.

6.1 Example of a Conditional

This 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` not only changes 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 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.

When the variable `CC` has the value '`gcc`', the above example has this effect:

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

When the variable `CC` has any other value, this effect is this:

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

Equivalent results can be obtained in another way by conditionalizing a variable assignment and then using the variable unconditionally:

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

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

6.2 Syntax of Conditionals

The syntax of a simple conditional with no `else` is as follows:

conditional-directive
text-if-true
`endif`

The *text-if-true* may be any lines of text, to be considered as part of the makefile if the condition is true. If the condition is false, no text is used instead.

The syntax of a complex conditional is as follows:

conditional-directive
text-if-true
`else`
text-if-false
`endif`

If the condition is true, *text-if-true* is used; otherwise, *text-if-false* is used instead. The *text-if-false* can be any number of lines of text.

Conditionals work at the textual level. The lines of the *text-if-true* are read as part of the makefile if the condition is true; if the condition is false, those lines are ignored 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.

You may use an `include` directive within a conditional, but you may not start a conditional in one file and end it in another.

The syntax of the *conditional-directive* is the same whether the conditional is simple or complex. There are four different directives that test different conditions. Here is a table of them:

`ifeq (`*arg1*`, `*arg2*`)`
: 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.

`ifneq (`*arg1*`, `*arg2*`)`
: Expand all variable references in *arg1* and *arg2* and compare them. If they are different, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

`ifdef` *variable-name*
: If the variable *variable-name* has a non-empty value, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective. Variables that have never been defined have an empty value.

`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.

6.3 Conditionals that Test Flags

You can write a conditional that tests `make` command flags such as '`-t`' by using the variable `MAKEFLAGS` together with the `findstring` function. This is useful when `touch` is not enough to make a file appear up to date.

The `findstring` function determines whether one string appears as a substring of another. If you want to test for the '`-t`' flag, use '`t`' as the first string and the value of `MAKEFLAGS` as the other.

For example, here is how to arrange to use '`ranlib -t`' to finish marking an archive file up to date:

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        @echo $(MAKE) > /dev/null
        touch archive.a
ranlib -t archive.a
else
        ranlib archive.a
endif
```

The `echo` command does nothing when executed; but its presence, with a reference to the variable `MAKE`, marks the rule as "recursive" so that its commands will be executed despite use of the '`-t`' flag.

7. 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.

7.1 Function Call Syntax

A function call resembles a variable reference. It looks like this:

`$(`*function arguments*`)`

or like this:

`${`*function arguments*`}`

Here *function* is a function name; one of a short list of names that are part of `make`. There is no provision for defining new functions.

The *arguments* are the arguments of the function. They are separated from the function name by one or more spaces and/or tabs, and if there are more than one argument they are separated by commas. Such whitespace and commas are not part of any argument's value. Parentheses or braces, whichever you use to surround the function call, can appear in an argument only in matching pairs; the ones that were not used to surround the function call can appear freely. If the arguments contain other function calls or variable references, it is wisest to surround them with the same delimiters used for the containing function call.

The text written for each argument is processed by substitution of variables and function calls in order to produce the argument value, which is the text on which the function acts.

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 those variables where such characters are wanted, like this:

```
comma= ,
```

```
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.

7.2 Functions for String Substitution and Analysis

Here are two functions that operate on substrings of a string: `subst` and `findstring`.

`$(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`’.

`$(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. See section 6.3 [Testing Flags], page 40, for a practical application of `findstring`.

Here is a realistic example of use of `subst`. Suppose that a makefile uses the `VPATH` variable to specify a list of directories that `make` should search for dependency files. This example shows how to tell the C compiler to search for header files in the same list of directories.

The value of `VPATH` is a list of directories separated by colons, such as ‘`src:../headers`’. First, the `subst` function is used to change the colons to spaces:

```
$(subst :, ,$(VPATH))
```

This produces ‘`src ../headers`’. Then another function, `addprefix`, can turn each directory name into an ‘`-I`’ flag. These can be added to the value of the variable `CFLAGS`, which is passed automatically to the C compiler, like this:

```
CFLAGS= $(CFLAGS) $(addprefix -I,$(subst :, ,$(VPATH)))
```

The effect is to append the text '`-Isrc -I../headers`' to the previously given value of `CFLAGS`.

7.3 Rescanning Text for Variable References

`$(expand` *text*`)`
: Expands the text *text* twice. This is to say, after variable references and function calls written in *text* are expanded, producing the argument text, that text is scanned over again for variable references and function calls.

For an example, suppose the following variable values have been set up:

```
foo=cross$$(intermediate)
intermediate=bar
```

where the double '`$`' is used in setting `foo` in order to prevent variable substitution from occurring when `foo` is set, thus getting an actual dollar sign into the value of `foo`.

Now if we expand `foo` in the usual way, with '`$(foo)`', the result is '`cross$(intermediate)`'.

The effect of '`$(expand $(foo))`' is to rescan that string '`cross$(intermediate)`', which appears as the expanded argument string. In the process, variable substitution operates on the variable `intermediate`. The ultimate result is '`crossbar`'.

The `expand` function is most often useful with canned sequences of commands (See section 4.6 [Sequences], page 29).

7.4 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 these functions performs a specific transformation on a file name. The argument of the function is regarded as a series of file names, separated by whitespace. (Leading and trailing whitespace is ignored.) Each file name in the series is transformed in the same way and the results are concatenated with single spaces between them.

`$(dir names)`

Extracts the directory-part of each file name in *names*. The directory-part of the file name is everything up through (and including) the last slash in it. If the file name contains no slash, the directory part is the string '`./`'. For example,

```
$(dir src/foo.c hacks)
```

produces the result '`src/ ./`'.

`$(notdir names)`

Extracts all but the directory-part of each file name in *names*. If the file name contains no slash, it is left unchanged. Otherwise, everything through the last slash is removed from it. A file name that ends with a slash becomes an empty string. This is unfortunate, because it means that the result does not always have the same number of whitespace-separated file names as the argument had; but we do not see any other valid alternative.

For example,

```
$(notdir src/foo.c hacks)
```

produces the result '`foo.c hacks`'.

`$(suffix names)`

Extracts the suffix of each file name in *names*. If the file name contains a period, the suffix is everything starting with the last period. Otherwise, the suffix is the empty string. This frequently means that the result will be empty when *names* is not, and if *names* contains multiple file names, the result may contain fewer file names.

For example,

```
$(notdir src/foo.c hacks)
```

produces the result '`.c`'.

`$(basename names)`

Extracts all but the suffix of each file name in *names*. If the file name contains a period, the basename is everything starting up to (and not including) the last period. Otherwise, the basename is the entire file name. For example,

```
$(basename src/foo.c hacks)
```

produces the result '`src/foo hacks`'.

`$(addsuffix suffix,names)`

The argument *names* is regarded as a series of names, separated by whitespace; *suffix* is used as a unit. The value of *suffix* is appended to the end of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addsuffix .c,foo bar)
```

produces the result '`foo.c bar.c`'.

`$(addprefix` *prefix*, *names*)

The argument *names* is regarded as a series of names, separated by whitespace; *prefix* is used as a unit. The value of *prefix* is appended to the front of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addprefix src/,foo bar)
```

produces the result '`src/foo src/bar`'.

`$(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`'.

`$(wildcard` *pattern*)

The argument *pattern* is a file name pattern, typically containing wildcards characters. The result of **`wildcard`** is a space-separated list of the names of existing files that match the pattern.

Wildcard are expanded automatically in rules (see section 3.2 [Wildcards], page 12). But it does not normally take place when a variable is set, or inside the arguments of other functions. Those occasions are when the **`wildcard`** function is useful.

8. 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. This is what `make` will do if run with no arguments.

But you might want to update only some of the files; you might want to use a different compiler or different compiler options; you might want just to find out which files are out of date without changing them.

By specifying arguments when you run `make`, you can do any of these things or many others.

8.1 Arguments to Specify the Makefile

The way to specify the name of the makefile is with the '`-f`' option. For example, '`-f altmake`' says to use the file '`altmake`' as the makefile.

If you use the '`-f`' flag several times (each time with a following argument), all the specified files are used jointly as makefiles.

If you do not use the '`-f`' flag, the default is to use '`./makefile`', or, if that does not exist, '`./Makefile`'. See chapter 2 [Makefiles], page 7.

8.2 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 or that appear in included makefiles). Therefore, makefiles are usually written so that the first target is for compiling the entire program or programs they describe.

You can specify a different goal or goal 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 '='). 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 a several programs, with a makefile that starts like this:

```
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 aren't 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 isn't a dependency of the default goal.

Another use of specifying a goal is to run the commands associated with a phony target (see section 3.4 [Phony Targets], page 16) or empty target (see section 3.5 [Empty Targets], page 18). 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`'. Here is a list of typical phony and empty target names:

'`clean`' Delete all files that the makefile could remake.

'`install`' Copy the executable file into a directory that users typically search for commands.

'`print`' Print listings of the source files that have changed.

'`tar`' Create a tar file of the source files.

8.3 Avoiding Recompilation of Some Files

Sometimes you may have changed a source file but you don't 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 don't need to be recompiled and you would rather not waste the time waiting.

If you anticipate the problem before making the change, you can use the '`-t`' flag. This flag tells `make` not to run the commands in the rules, but rather to mark the target up-to-date by changing its last-modification date. You would follow this procedure:

1. Use the command '`make`' to recompile the source files that really need recompilation.
2. Make the changes in the header files.
3. Use the command '`make -t`' to mark all the object files as up-to-date. The next time you run `make`, the changes in the header files will not cause any recompilation.

If you have already changed the header file at a time when some files do need recompilation, it is too late to do this. Instead, you can use the '`-o` *file*' flag, which marks a specified file as "old" (see section 8.7 [Options], page 51). This means that the file itself won't be remade, and nothing else will be remade on its account. Follow this procedure:

1. Recompile the source files that need compilation for reasons independent of the particular header file, with '`make -o` *headerfile*'. If several header files are involved, use a separate '`-o`' option for each header file.
2. Touch all the object files with '`make -t`'.

8.4 Instead of Executing the Commands

The makefile tells `make` how to tell whether a target is up to date, and how to update each target. But updating the targets is not always what you want. Certain options specify other activities for `make`.

'`-t`' "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.

'`-n`' "No-op". The activity is to print what commands would be used to make the targets up to date, but not actually execute them.

'`-q`' "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.

With the '`-n`' flag, `make` prints without execution the commands that it would normally execute.

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.

It is an error to use more than one of these three flags in the same invocation of `make`.

8.5 Overriding Variables

You can override the value of a variable using an arguments to `make` that contains a '`=`'. The argument '*v*=*x*' sets the value of the variable *v* to *x*.

Values specified this way override all values specified in the makefile itself; once you have set a variable with a command argument, any ordinary attempt in the makefile to change that variable is simply ignored.

One 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 like this:

```
cc -c $(CFLAGS) foo.c
```

Thus, whatever value you set for `CFLAGS` affects each compilation that occurs. The makefile probably specifies the usual value for `CFLAGS`, like this:

```
CFLAGS=-g
```

Each time you run `make`, you can override this value and specify a different value. 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 enclose spaces and other special characters in the value of a variable when you override it.)

The variable `CFLAGS` is only one of many standard variables that exist just so that you can change them this way. See section 9.3 [Implicit Variables], page 58, for a complete list.

You can also program the makefile to look at additional variables of your own, giving the user ability to control other aspects of how the makefile works by changing the variables.

There is one way that the makefile can change a variable that you have overridden. This is to use the `override` directive, which is a line that looks like this:

```
override variable = value
```

This line acts like an ordinary variable assignment except that it is not ignored even if you have used a commadn option to set the variable. See section 5.4 [Override Directive], page 33.

8.6 Testing the Compilation of a Program

Normally, when an error happens in executing a shell command, `make` gives up immediately, returning a nonzero status. No further commands are executed for any target. The error implies that the goal cannot be correctly remade, so `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 all the compilation errors.

Then you should use the '`-k`' flag. If the '`-k`' flag is specified, `make` continues to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, '`make -k`' will continue compiling other object files even though it already knows that linking them will be impossible. See section 8.7 [Options], page 51.

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's `compile` command passes the '`-k`' flag by default.

8.7 Summary of Options

Here is a table of all the options `make` understands:

'-b' This option is ignored for compatibility with other versions of **make**.

'-c *dir*' Change to directory *dir* before executing the rules. If multiple '-c' options are specified, each is interpreted relative to the previous one: '-c / -c etc' is equivalent to '-c /etc'. This is typically used with recursive invocations of **make** (see section 4.5 [Recursion], page 26).

'-d' 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.

'-f *file*' Use file *file* as a makefile. See chapter 2 [Makefiles], page 7.

'-i' Ignore all errors in commands executed to remake files. See section 4.3 [Errors], page 24.

'-I *dir*' Specifies a directory *dir* to search for included makefiles. See section 2.3 [Include], page 8. If several '-I' options are used to specify several directories, the directories are searched in the order specified.

'-k' Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same. See section 8.6 [Testing], page 51.

'-n' Print the commands that would be executed, but do not execute them. See section 8.4 [Instead of Execution], page 49.

'-o *file*' Do not remake the file *file* even if it is older than its dependencies, and do not remake anything on account of changes in *file*. Essentially the file is treated as very old and its rules are ignored. See section 8.3 [Avoid Compilation], page 48.

'-p' Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as otherwise specified.

'-q' "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, nonzero otherwise. See section 8.4 [Instead of Execution], page 49.

'-r' Eliminate use of the built-in implicit rules (see chapter 9 [Implicit], page 55). Also clear out the default list of suffixes for suffix rules (see section 9.7 [Suffix Rules], page 66).

'-s' Silent operation; do not print the commands as they are executed. See section 4.1 [Echoing], page 23.

'-S' Cancel the effect of the '-k' option. This is never necessary except in a recursive **make** where '-k' might be inherited from the top-level **make** via **MAKEFLAGS**. See section 4.5 [Recursion], page 26.

'-t' Touch files (mark them up to date without really changing them) instead of running their commands. This is used to pretend (to fool future invocations of **make**) that the commands were done. See section 8.4 [Instead of Execution], page 49.

‘-w’ Print a message containing the working directory both before and after executing the makefile; this is useful for tracking down errors from builds of large directory trees. See section 4.5 [Recursion], page 26.

9. Using Implicit Rules

Certain standard ways of remaking target files are used very often. For example, one customary way to make an object file is from a C source file using the C compiler, `cc`.

Implicit rules tell `make` how to use customary techniques so that you don't have to specify them in detail when you want to use them. For example, there is an implicit rule for C compilation.

Implicit rules work based on file names. For example, C compilation typically takes a '`.c`' file and makes a '`.o`' file. So `make` applies the implicit rule when it sees this combination of file-name endings.

A chain of implicit rules can apply in sequence; for example, `make` will remake a '`.o`' file from a '`.y`' file by way of a '`.c`' file. See section 9.4 [Chained Rules], page 60.

The built-in implicit rules use several variables in their commands so that, by changing the values of the variables, you can change the way the implicit rule works. For example, the variable `CFLAGS` controls the flags given to the C compiler by the implicit rule for C compilation. See section 9.3 [Implicit Variables], page 58.

You can define your own implicit rules by writing *pattern rules*. See section 9.5 [Pattern Rules], page 61.

9.1 Using Implicit Rules

To allow `make` to find a customary method for updating a target file, all you have to do is refrain from specifying commands yourself. Either write a rule with no command lines, or don't write a rule at all. Then `make` will figure out which implicit rule to use based on which kind of source file exists.

For example, suppose the makefile looks like this:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Because you mention '`foo.o`' but do not give a rule for it, `make` will automatically look for an implicit rule that tells how to update it. This happens whether or not the file '`foo.o`' currently exists.

If an implicit rule is found, it supplies both commands and a dependency (the source file). You would want to write a rule for '`foo.o`' with no command lines if you need to specify additional dependencies, such as header files, that the implicit rule cannot supply.

Each implicit rule has a target pattern and dependency patterns. There may be many implicit rules with the same target pattern. For example, numerous rules make '`.o`' files: one, from a '`.c`' file with the C compiler; another, from a '`.p`' file with the Pascal compiler; and so on. The rule that actually applies is the one whose dependency exists 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 other files are supposed to exist. See section 9.2 [Catalogue of Rules], page 56, for a catalogue of all the predefined implicit rules.

Above, we said an implicit rule applies if the required dependency "exists 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 the implicit dependency is the result of another implicit rule, we say that *chaining* is occurring. See section 9.4 [Chained Rules], page 60.

In general, `make` searches for an implicit rule for each target, and for each double-colon rule, that has no commands. A file that is mentioned only as a dependency is considered a target whose rule specifies nothing, so implicit rule search happens for it. See section 9.8 [Search Algorithm], page 67, for the details of how the search is done.

9.2 Catalogue of Implicit Rules

Here is a catalogue of predefined implicit rules which are always available unless the makefile explicitly overrides or cancels them. (See section 9.5.5 [Cancelling Rules], page 65, for information on cancelling or overriding an implicit rule. The '`-r`' option cancels all predefined rules.)

Compiling C programs

'`n.o`' will be made automatically from '`n.c`' with the command '`$(CC) -c $(CFLAGS)`'.

Compiling Pascal programs
: '*n*.o' will be made automatically from '*n*.p' with the command '`$(PC) -c $(PFLAGS)`'.

Compiling Fortran, EFL and Ratfor programs
: '*n*.o' will be made automatically from '*n*.e', '*n*.r', '*n*.F' or '*n*.f' by running the Fortran compiler. The precise command used is as follows:

  '.e' '`$(FC) -c $(EFLAGS)`'.

  '.f' '`$(FC) -c $(FFLAGS)`'.

  '.F' '`$(FC) -c $(FFLAGS)`'.

  '.r' '`$(FC) -c $(RFLAGS)`'.

Preprocessing Fortran, EFL and Ratfor programs
: '*n*.f' will be made automatically from '*n*.e', '*n*.r' or '*n*.F'. This rule runs just the preprocessor to convert a Ratfor, EFL or preprocessable Fortran program into a strict Fortran program. The precise command used is as follows:

  '.e' '`$(FC) -F $(EFLAGS)`'.

  '.F' '`$(FC) -F $(FFLAGS)`'.

  '.r' '`$(FC) -F $(RFLAGS)`'.

Assembling assembler programs
: '*n*.o' will be made automatically from '*n*.s' by running the assembler `as`. The precise command used is '`$(AS) $(ASFLAGS)`'.

Linking a single object file
: '*n*' will be made automatically from '*n*.o' by running the linker `ld` via the C compiler. The precise command used is '`$(CC) $(LDFLAGS) … $(LOADLIBES)`'.

  This rule does the right thing for a simple program with only one source file. In more complicated cases, you must write an explicit command for linking.

Compiling C into assembler code
: '*n*.s' will be made automatically from '*n*.c' with the command '`$(CC) -S $(CFLAGS)`'.

  It would be possible for `make` to convert '*n*.c' into '*n*.o' by way of '*n*.s', using this rule and the rule for running the assembler. But that is not what `make` does, because the rule for compiling '*n*.c' into '*n*.o' directly comes earlier in the order of rules. The upshot is that the file '*n*.s' is not created or changed when '*n*.o' is being remade. This rule is used only if you explicitly specify '*n*.s' as a goal or needed dependency.

  This is a deliberate decision, for the sake of compatibility with Unix `make`.

  If you want `make` update '*n*.s' on the way to updating '*n*.o', you can request this by cancelling the other rule that allows direct compilation. See section 9.5.5 [Cancelling Rules], page 65.

Compiling Pascal, Fortran, EFL or Ratfor into assembler code
: '*n*.s' will be made automatically from '*n*.p', '*n*.e', '*n*.r', '*n*.F' or '*n*.F' by running the appropriate compiler with the '`-S`' flag instead of the '`-c`' flag.

For compatibility with Unix `make`, these rules apply only if you expressly request `make` to update '*n*.`s`'. See the information immediately above.

Yacc for C programs
: '*n*.`c`' will be made automatically from '*n*.`y`' by running Yacc with the command '`$(YACC) $(YFLAGS)`'.

Yacc for Ratfor programs
: '*n*.`r`' will be made automatically from '*n*.`yr`' by running Yacc with the command '`$(YACCR) $(YFLAGS)`'.

Yacc for EFL programs
: '*n*.`e`' will be made automatically from '*n*.`ye`' by running Yacc with the command '`$(YACCE) $(YFLAGS)`'.

Lex for C programs
: '*n*.`c`' will be made automatically from '*n*.`l`' by by running Lex. The actual command is '`$(LEX) $(LFLAGS)`'.

Lex for Ratfor programs
: '*n*.`r`' will be made automatically from '*n*.`l`' by by running Lex. The actual command is '`$(LEX) $(LFLAGS)`'.

The traditional custom of using the same suffix '`.l`' for all Lex files regardless of whether they produce C code or Ratfor code makes it impossible for `make` to determine autmatically 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.

RCS
: Any file '*n*' will be extracted if necessary from an RCS file named either '*n*`,v`' or '`RCS/`*n*`,v`'. The precise command used is '`$(CO) $(COFLAGS)`'. The variable `CO` has default value '`co`'.

SCCS
: Any file '*n*' will be extracted if necessary from an SCCS file named either '`s.`*n*' or '`SCCS/s.`*n*'. The precise command used is '`$(GET) $(GFLAGS)`'.

I recommend that you avoid the use of SCCS. RCS is widely held to be superior, and RCS is also free. By choosing free software in place of comparable proprietary software, you support the free software movement.

9.3 Variables Used by Implicit Rules

The commands in built-in implicit rules make liberal use of certain predefined variables. You can redefine these variables, either in the makefile or with arguments to `make`, to alter how the implicit rules work without actually redefining them.

For example, the command used to compile a C source file actually says '`$(CC) -c $(CFLAGS)`'. The default values of the variables used are '`cc`' and nothing, resulting in the command '`cc -c`'. By redefining '`$(CC)`' to '`ncc`', you could cause '`ncc`' to be used for all C compilations performed by the implicit rule. By redefining '`$(CFLAGS)`' to be '`-g`', you could pass the '`-g`' option to each compilation. *All* implicit rules that do C compilation use '`$(CC)`' to get the program name for the compiler and *all* include '`$(CFLAGS)`' among the arguments given to the compiler.

The variables used in implicit rules fall into two classes: those that are names of programs (like `CC`) and those that contain arguments for the programs (like `CFLAGS`). (The "name of a program" may also contain some command arguments, but it must start with an actual executable program name.) If a variable value contains more than one argument, separate them with spaces.

Here is a table of variables used as names of programs:

`AS` Program for doing assembly; default '`as`'.

`CC` Program for compiling C programs; default '`cc`'.

`CO` Program for extracting a file from RCS; default '`co`'.

`FC` Program for compiling or preprocessing Fortran programs (or Ratfor or EFL 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`'.

`YACCE` Program to use to turn Yacc grammars into EFL programs; default '`yacc -e`'.

`RANLIB` Program to use to update the symbol-directory of an archive (the '`__.SYMDEF`' member); default '`ranlib`'.

Here is a table of variables whose values are additional arguments for the programs above:

`ASFLAGS` Extra flags to give to the assembler (when explicitly invoked on a '`.s`' file).

`CFLAGS` Extra flags to give to the C compiler; default is empty.

`EFLAGS` Extra flags to give to the Fortran compiler for EFL programs; default is empty.

`FFLAGS` Extra flags to give to the Fortran compiler; default is empty.

`LFLAGS` Extra flags to give to Lex; default is empty.

`LDFLAGS` Extra flags to give to compilers when they are supposed to invoke the linker, '`ld`'; default is empty.

`PFLAGS` Extra flags to give to the Pascal compiler; default is empty.

`RFLAGS` Extra flags to give to the Fortran compiler for Ratfor programs; default is empty.

`YFLAGS` Extra flags to give to Yacc; default is empty.

9.4 Chains of Implicit Rules

Sometimes a file can be made by a sequence of implicit rules. For example, a file '*n*`.o`' could be made from '*n*`.y`' by running first Yacc and then `cc`. Such a sequence is called a *chain*.

If the file '*n*`.c`' exists, or is mentioned in the makefile, no special searching is required: **`make`** finds that the object file can be made by C compilation from '*n*`.c`'; later on, when considering how to make '*n*`.c`', the rule for running Yacc will be 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 optionally define an implicit rule so as to preserve certain intermediate files.)

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 by handled by rule chains. For example, making '`foo`' from '`foo.c`' could be handled by compiling and linking with separate 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.

9.5 Defining and Redefining Pattern Rules

You define an implicit rule by writing a *pattern rule*. A pattern rule looks like an ordinary rule, except that its target contains the character '%' (exactly one of them). The target is considered a pattern for matching file names; the '%' can match any 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*.

A pattern rule must have at least one dependency that uses '%'. '%' 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.

There may also be dependencies that do not use '%'; such a dependency attaches to every file made by this pattern rule. These unvarying dependencies are rarely useful.

The order in which pattern rules appear in the makefile is important because the rules are considered in that order. Of equally applicable rules, the first one found is used. The rules you write take precedence over those that are built in. Note, however, that a rule whose dependencies actually exist or are mentioned always takes priority over a rule with dependencies that must be made by chaining other implicit rules.

9.5.1 Pattern Rule Examples

Here are some examples of pattern rules actually predefined in **make**. First, the rule that compiles '.c' files into '.o' files:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $< -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 as they are in each case where the rule apply (see section 9.5.2 [Automatic], page 62).

Here is a second built-in rule:

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

defines a rule that can make any file ‘*x*’ whatever from a corresponding file ‘x,v’ in the subdirectory ‘RCS’. Since the target is ‘%’, this rule will apply to any file whatever, provided the appropriate dependency file exists. The double colon makes the rule *terminal*, which means that its dependency may not be an intermediate file (see section 9.5.4 [Match-Anything Rules], page 64).

9.5.2 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 can't write the name in the command, because the name is different each time the implicit rule is applied.

What you do is use a special feature of **make**, the *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. In this example, you would use ‘$@’ for the object file name and ‘$<’ for the source file name.

Here is a table of automatic variables:

**$@** The file name of the target of the rule. If the target is an archive member, then ‘$@’ is the name of the archive file.

$% The target member name, when the target is an archive member. For example, if the target is ‘foo.a(bar.o)’ then ‘$%’ is ‘bar.o’ and ‘$@’ is ‘foo.a’. ‘$%’ is empty when the target is not an archive member.

$< The name of the first implicit dependency, when an implicit rule is being applied.

$? The names of all the dependencies that are newer than the target, with spaces between them.

$^ The names of all the dependencies, with spaces between them.

$* The stem with which an implicit rule matches (see section 9.5.3 [Pattern Match], page 63). 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.

`$($@)` The same as `$@`, for compatibility with some other versions of `make`.

'`$?`' 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 c lib $?
```

Of the variables listed above, four have values that are single file names. These four 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 an equivalent effect (see section 7.4 [Filename Functions], page 43). Here is a table of the variants:

'`$(@D)`' The directory part of the file name of the target. If the value of '`$@`' is '`dir/foo.o`' then '`$(@D)`' is '`dir/`'. This value is '`./`' if '`$@`' does not contain a slash. '`$(@D)`' is equivalent to '`$(dir $@)`'.

'`$(@F)`' The file-within-directory part of the file name of the target. If the value of '`$@`' is '`dir/foo.o`' then '`$(@F)`' is '`foo.o`'. '`$(@F)`' is equivalent to '`$(notdir $@)`'.

'`$($/)`' The same as `$(@F)`, for compatibility with some other versions of `make`.

'`$(%D)`'
'`$(%F)`' The directory part and the file-within-directory part of the archive member name.

'`$(*D)`'
'`$(*F)`' The directory part and the file-within-directory part of the stem; '`dir/`' in this example.

'`$(<D)`'
'`$(<F)`' The directory part and the file-within-directory part of the first implicit dependency.

9.5.3 How Patterns Match

A target pattern is composed of a '`%`' between a prefix and a suffix, either 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 usually it does not), directory names in the file names are removed from the file name before it is compared with the target prefix and suffix. The directory names, along with the slash that ends them, are added back to the stem. Thus, '`e%t`' does match 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`'.

9.5.4 Match-Anything Pattern Rules

When a pattern rule's target is just '`%`', it matches any filename 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`', '`foo.c.p`', etc. 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.

For example, the built-in implicit rules for extracting sources from RCS and SCCS files are terminal; as a result, if the file '`foo.c,v`' does not exist, `make` will not even consider trying to make it as an intermediate file from '`foo.c,v.o`' or from '`RCS/SCCS/s.foo.c,v`'. RCS and SCCS files are generally ultimate source files, which should not be remade from any other files; therefore, `make` can save time by not looking for ways to remake them.

If you do not mark the match-anything rule as terminal, then it is nonterminal. A nonterminal match-anything rule cannot apply to a file name that indicates a specific type of data. A file name indicates a specific type of data if some non-match-anything implicit rule target matches it.

For example, the file name '`foo.c`' matches the target for the pattern rule '`%.c : %.y`' (the rule to run Yacc). Regardless of whether this rule is actually applicable (which happens only if there is a file '`foo.y`'), the fact that its target matches is enough to prevent consideration of any nonterminal match-everything 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-everything rules are used for making files containing specific types of data (such as executable files) and a file name with a recognized suffix indicates a specific different 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-everything rules won't 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`'.

9.5.5 Cancelling Implicit Rules

You can override a built-in implicit rule 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
```

9.6 Defining Last-Resort Default Rules

You can define a last-resort implicit rule by writing a rule for the target `.DEFAULT`. 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. Naturally, there is no `.DEFAULT` rule unless you write one.

For example, when testing a makefile, you might not care if the source files contain real data, only that they exist. Then you might do this:

```
.DEFAULT:
        touch $@
```

to cause all the source files needed (as dependencies) to be created silently.

9.7 Old-Fashioned Suffix Rules

Suffix rules are the old-fashioned way of defining implicit rules for `make`. Suffix rules are obsolete because pattern rules are more general and clearer. They are supported in GNU `make` for compatibility with old makefiles. They come in two kinds: *double-suffix* and *single-suffix*.

A double-suffix rule is defined by a pair of suffixes: the target suffix and the source suffix. It matches any file whose name ends with the target suffix. The corresponding implicit dependency is to the file name made by replacing the target suffix with the source suffix. A two-suffix rule whose target and source suffixes are '`.o`' and '`.c`' is equivalent to the pattern rule '`%.o : %.c`'.

A single-suffix rule is defined by a single suffix, which is the source suffix. It matches any file name, and the corresponding implicit dependency name is made by appending the source suffix. A single-suffix rule whose source suffix is '`.c`' is equivalent to the pattern rule '`% : %.c`'.

Suffix rule definitions are recognized by comparing each rule's target against a defined list of known suffixes. When `make` sees a rule whose target is a known suffix, this rule is considered a single-suffix rule. When `make` sees a rule whose target is two known suffixes concatenated, this rule is taken as a double-suffix rule.

For example, '`.c`' and '`.o`' are both on the default list of known suffixes. Therefore, if you define a rule whose target is '`.c.o`', `make` takes it to be a double-suffix rule with source suffix '`.c`' and target suffix '`.o`'. For example, here is the old fashioned way to define the rule for compiling a C source:

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

The known suffixes are simply the names of the dependencies of the special target `.SUFFIXES`. You can add your own suffixes by writing a rule for `.SUFFIXES` that adds more dependencies, as in

```
.SUFFIXES: .hack .win
```

which adds '`.hack`' and '`.win`' to the end of the list of suffixes.

If you wish to eliminate the default known suffixes instead of just adding to them, write a rule for `.SUFFIXES` with no dependencies. By special dispensation, this eliminates all existing dependencies of `.SUFFIXES`. You can then write another rule to add the suffixes you want. For example,

```
.SUFFIXES:            # Delete the default suffixes
.SUFFIXES: .c .o .h   # Define our suffix list
```

The '`-r`' flag causes the default list of suffixes to be empty.

9.8 Implicit Rule Search Algorithm

Here is the procedure `make` uses for searching for an implicit rule for a target *t*. This procedure is followed for each double-colon rule with no commands, for each target of ordinary rules none of which have commands, and for each dependency that is not the target of any rule. It is also followed recursively for dependencies that come from implicit rules, in the search for a chain of rules.

Suffix rules are not mentioned in this algorithm because suffix rules are converted to equivalent pattern rules after the makefiles have been read in.

For an archive member target of the form '*archive*(*member*)', the following algorithm is run twice, first using '(*member*)' as the target *t*, and second using the entire target 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 the pattern rules whose target matches *d* or *n*. If the target pattern contains a slash, it is matched against *t*; otherwise, against *n*.
3. If any rule in that list is *not* a match-anything rule, then remove all nonterminal match-anything rules from the list.
4. Remove any rules with no dependencies from the list.
5. For each pattern rule in the list:
   1. Find the stem *s*: the part of *d* or *n* that the '`%`' in the target pattern matches.
   2. Compute the dependency names by substituting *s* for '`%`'; if the target pattern does not contain a slash, *d* is appended to the front of each dependency name.
   3. Test whether all the dependencies exist or ought to exist. (If a file name 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, then this rule applies.
6. If no pattern rule has been found so far, try harder. For each pattern rule in the list:
   1. If the rule is a terminal match-anything rule, ignore it and go on to the next rule.
   2. Compute the dependency names as before.
   3. Test whether all the dependencies exist or ought to exist.
   4. For each dependency that does not exist, follow this algorithm recursively to see if the dependency can be made by an implicit rule.
   5. If all dependencies exist, ought to exist, or can be made by implicit rules, then this rule applies.
7. If no rule has been found so far, this target cannot be made by an implicit rule. Return failure.
8. 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*.

When the commands of a pattern rule are executed for *t*, the automatic variables '`$@`', '`$*`' and '`$<`' are set as follows:

'`$@`'
: *t*

'`$*`'
: If the target pattern contains a slash, this is *s*; otherwise, it is *ds*.

'`$<`'
: The name of the first dependency that came via the implicit rule.

For `.DEFAULT` commands, as for non-implicit commands, '`$*`' and '`$<`' are empty. '`$@`' is *t*, as always.

10. 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.

10.1 Archive Members as Targets

An individual member of an archive file can be used as a target or dependency in `make`. The archive file must already exist, but the member need not exist. You specify the member named *member* in archive file *archive* as follows:

archive(*member*)

This construct is available only in targets and dependencies, not in commands! Most programs that you might use in commands do not support this syntax and cannot act directly on archive members. Only `ar` and other programs specifically designed to operate on archives can do so. Therefore, valid commands to update an archive member target probably must use `ar`. For example, this rule says to create a member '`hack.o`' in archive '`foolib`' by copying the file '`hack.o`':

```
foolib(hack.o) : hack.o
        ar r 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.

10.2 Implicit Rule for Archive Member Targets

Recall that a target that looks like '*a*(*m*)' stands for the member named *m* in the archive file *a*.

When `make` looks for an implicit rule for such a target, as a special feature it considers implicit rules that match '(*m*)', as well as those that match the actual target '*a*(*m*)'.

This causes one special rule whose target is '`(%)`' to match. This rule updates the target '*a*(*m*)' by copying the file *m* into the archive. For example, it will update the archive member target '`foo.a(bar.o)`' by copying the *file* '`bar.o`' into the archive '`foo.a`' as a member named '`bar.o`'.

When this rule is chained with others, the result is very powerful. The command '`make "foo.a(bar.o)"`' 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 the file '`bar.o`' has been invented as an intermediate file.

10.2.1 Updating Archive Symbol Directories

An archive file that is used as a library usually contains a special member named '`__.SYMDEF`' that contains a directory of the external symbol names defined by all the other members. After you update any other members, you need to update '`__.SYMDEF`' so that it will summarize the other members properly. This is done by running the `ranlib` program:

`ranlib` *archivefile*

Normally you would put this command in the rule for the archive file, and make all the members of the archive file dependents of that rule. For example,

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

The effect of this is to update archive members '`x.o`', '`y.o`', etc., and then update the symbol directory member '`__.SYMDEF`' by running `ranlib`. The rules for updating the members are not shown here; most likely you can omit them and use the implicit rule which copies files into the archive, as described in the preceding section.

11. Missing Features in GNU `make`

The `make` programs in various other systems support two features that are not implemented in GNU `make`.

- A target of the form '*file*((*entry*))' stands for a member of archive file *file*. The member is chosen, not by name, but by being an object file which defines the linker symbol *entry*.

  This feature was not put into GNU `make` because of the nonmodularity of putting knowledge into `make` of the internal format of archive file symbol directories. See section 10.2.1 [Archive Symbols], page 70.
- Suffixes (used in suffix rules) that end with the character '~' have a special meaning; 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 files '*n*`.o`' file from the SCCS file '`s.`*n*`.c`'. For complete coverage, a whole series of such suffix rules is required. See section 9.7 [Suffix Rules], page 66.

  In GNU `make`, this entire series of cases is handled by two pattern rules for extraction from SCCS, in combination with the general feature of rule chaining. See section 9.4 [Chained Rules], page 60.

Table of Contents