4th Edition

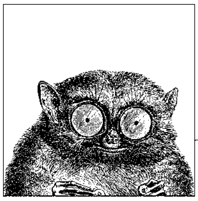
UNIX IN A NUTSHELL

A Desktop Quick Reference

Covers GNU/Linux, Mac OS X, and Solaris

O'REILLY®

Arnold Robbins



16

The GNU make Utility

The `make` program is a long time mainstay of the Unix toolset. It automates the building of software and documentation based on a specification of dependencies among files; e.g., object files that depend upon program source files, or PDF files that depend upon documentation program input files. GNU `make` is the standard version for GNU/Linux and Mac OS X.

This chapter presents the following topics:

- Conceptual overview
- Command-line syntax
- Makefile lines
- Macros
- Special target names
- Writing command lines

For more information, see *Managing Projects with GNU make* and *GNU Make: A Program for Directing Recompilation*, both listed in the Bibliography.

The software download site for GNU `make` is *ftp://ftp.gnu.org/gnu/make/*.

Conceptual Overview

The `make` program generates a sequence of commands for execution by the Unix shell. It uses a table of file dependencies provided by the programmer, and with this information, can perform updating tasks automatically for the user. It can keep track of the sequence of commands that create certain files, and the list of files or programs that require other files to be current before they can be rebuilt correctly. When a program is changed, `make` can create the proper files with a minimum of effort.

Each statement of a dependency is called a *rule*. Rules define one or more *targets*, which are the files to be generated, and the files they depend upon, the *prerequisites* or *dependencies*. For example, `prog.o` would be a target that depends upon `prog.c`; each time you update `prog.c`, `prog.o` must be regenerated. It is this task that `make` automates, and it is a critical one for large programs that have many pieces.

The file containing all the rules is termed a *makefile*; for GNU `make`, it may be named `GNUmakefile`, `makefile` or `Makefile`, in which case `make` will read it automatically, or you may use a file with a different name and tell `make` about it with the `-f` option.

Over the years, different enhancements to `make` have been made by many vendors, often in incompatible ways. POSIX standardizes how `make` is supposed to work. Today, GNU `make` is the most popular version in the Unix world. It has (or can emulate) the features of just about every other version of `make`, and many Open Source programs require it.

This chapter covers GNU `make`. Commercial Unix systems come with versions derived from the original System V version; these can be used for bootstrapping GNU `make` if need be. On the x86 versions of Solaris 10, you can find GNU `make` in `/usr/sfw/bin/gmake`. It isn't available on the Sparc version, although it can be easily bootstrapped with the standard version of `make` in `/usr/ccs/bin`.

Command-Line Syntax

The `make` program is invoked as follows:

```
make [options] [targets] [macro definitions]
```

Options, targets, and macro definitions can appear in any order. The last assignment to a variable is the one that's used. Macro definitions are typed as:

```
name=string
```

or

```
name:=string
```

For more information, see the section "Creating and Using Macros," later in this chapter.

If no `GNUmakefile`, `makefile`, or `Makefile` exists, `make` attempts to extract the most recent version of one from either an RCS file, if one exists, or from an SCCS file, if one exists. Note though, that if a real makefile exists, `make` will not attempt to extract one from RCS or SCCS, even if the RCS or SCCS file is newer than the makefile.

Options

Like just about every other GNU program, GNU `make` has both long and short options. The available options are as follows:

`-b` Silently accepted, but ignored, for compatibility with other versions of `make`.

`-B, --always-make`
: Treat all targets as out of date. All targets are remade, no matter what the actual status is of their prerequisites.

`-C` *`dir`*, `--directory=`*`dir`*
: Change directory to *dir* before reading makefiles. With multiple options, each one is relative to the previous. This is usually used for recursive invocations of `make`.

`-d`
: Print debugging information in addition to regular output. This information includes which files are out of date, the file times being compared, the rules being used to update the targets, and so on. Equivalent to `--debug=a`.

`--debug[=`*`debug-opt`*`]`
: Print debugging information as specified by *debug-opt*, which is one or more of the following letters, separated by spaces or commas. With no argument, provide basic debugging.

`a`	All. Enable all debugging.
`b`	Basic. Print each target that is out of date, and whether or not the build was successful.
`i`	Implicit. Like basic, but include information about the implicit rules searched for each target.
`j`	Jobs. Provide information about subcommand invocation.
`m`	Makefiles. Enable basic debugging, and any of the other options, for description of attempts to rebuild makefiles. (Normally, `make` doesn't print information about its attempts to rebuild makefiles.)
`v`	Verbose. Like basic, but also print information about which makefiles were read, and which prerequisites did not need to be rebuilt.

`-e`, `--environment-overrides`
: Environment variables override any macros defined in makefiles.

`-f` *`file`*, `--file=`*`file`*, `--makefile=`*`file`*
: Use *file* as the makefile; a filename of - denotes standard input. `-f` can be used more than once to concatenate multiple makefiles. With no `-f` option, `make` first looks for a file named `GNUmakefile`, then one named `makefile`, and finally one named `Makefile`.

`-h`, `--help`
: Print a usage summary, and then exit.

`-i`, `--ignore-errors`
: Ignore error codes from commands (same as `.IGNORE`).

`-I` *`dir`*, `--include-dir=`*`dir`*
: Look in *dir* for makefiles included with the `include` directive. Multiple options add more directories to the list; `make` searches them in order.

`-j [`*`count`*`]`, `--jobs[=`*`count`*`]`
: Run commands in parallel. With no *count*, `make` runs as many separate commands as possible. (In other words, it will build all the targets that are independent of each other, in parallel.) Otherwise, it runs no more than *count* jobs. This can decrease the time it takes to rebuild a large project.

`-k`, `--keep-going`
: Abandon the current target when it fails, but keep working with unrelated targets. In other words, rebuild as much as possible.

`-l` [*load*], `--load-average`[=*load*], `--max-load`[=*load*]
: If there are jobs running and the system load average is at least *load*, don't start any new jobs running. Without an argument, clear a previous limit. The *load* value is a floating point number.

`-m` Silently accepted, but ignored, for compatibility with other versions of `make`.

`-n`, `--dry-run`, `--just-print`, `--recon`
: Print commands but don't execute (used for testing). `-n` prints commands even if they begin with `@` in the makefile.

  Lines that contain `$(MAKE)` are an exception. Such lines *are* executed. However, since the `-n` is passed to the subsequent `make` in the MAKEFLAGS environment variable, that `make` also just prints the commands it executes. This allows you to test out all the makefiles in a whole software hierarchy without actually doing anything.

`--no-print-directory`
: Don't print the working directory as `make` runs recursive invocations. Useful if `-w` is automatically in effect but you don't want to see the extra messages.

`-o` *file*, `--assume-old=`*file*, `--old-file=`*file*
: Pretend that *file* is older than the files that depend upon it, even if it's not. This avoids remaking the other files that depend on *file*. Use this in cases where you know that the changed contents of *file* will have no effect upon the files that depend upon it; e.g., changing a comment in a header file.

`-p`, `--print-data-base`
: Print macro definitions, suffixes, and built-in rules. In a directory without a makefile, use `env -i make -p` to print out the default variable definitions and built-in rules.

`-q`, `--question`
: Query; return 0 if the target is up to date; nonzero otherwise.

`-r`, `--no-builtin-rules`
: Do not use the default rules. This also clears out the default list of suffixes and suffix rules.

`-s`, `--quiet`, `--silent`
: Do not display command lines (same as `.SILENT`).

`-S`, `--no-keep-going`, `--stop`
: Cancel the effect of a previous `-k`. This is only needed for recursive `make` invocations, where the `-k` option might be inherited via the MAKEFLAGS environment variable.

`-t`, `--touch`
: Touch the target files, causing them to be updated.

`-v`, `--version`
: Print version, copyright, and author information, and exit.

`-w`, `--print-directory`
: Print the working directory, before and after executing the makefile. Useful for recursive `make` invocations. This is usually done by default, so it's rare to explicitly need this option.

`--warn-undefined-variables`
: Print a warning message whenever an undefined variable is used. This is useful for debugging complicated makefiles.

`-W` *`file`*, `--assume-new=`*`file`*, `--new-file=`*`file`*, `--what-if=`*`file`*
: Treat *file* as if it had just been modified. Together with `-n`, this lets you see what `make` would do if *file* were modified, without actually doing anything. Without `-n`, `make` pretends that the file is freshly updated, and acts accordingly.

Makefile Lines

Instructions in the makefile are interpreted as single lines. If an instruction must span more than one input line, use a backslash (\) at the end of the line so that the next line is considered a continuation. The makefile may contain any of the following types of lines:

Blank lines
: Blank lines are ignored.

Comment lines
: A number sign (#) can be used at the beginning of a line or anywhere in the middle. `make` ignores everything after the #.

Dependency lines
: One or more target names, a single- or double-colon separator, and zero or more prerequisites:

```
targets : prerequisites
targets :: prerequisites
```

In the first form, subsequent commands are executed if the prerequisites are newer than the target. The second form is a variant that lets you specify the same targets on more than one dependency line. (This second form is useful when the way you rebuild the target depends upon which prerequisite is newer.) In both forms, if no prerequisites are supplied, subsequent commands are always executed (whenever any of the targets are specified). For example, the following is invalid, since single-colon rules do not allow targets to repeated:

```
# PROBLEM: Single colon rules disallow repeating targets
whizprog.o: foo.h
        $(CC) -c $(CFLAGS) whizprog.o
        @echo built for foo.h

whizprog.o: bar.h
        $(CC) -c $(CFLAGS) whizprog.o
        @echo built for bar.h
```

In such a case, the last set of rules is used and make issues a diagnostic. However, double-colon rules treat the dependencies separately, running each set of rules if the target is out of date with respect to the individual dependencies:

```
# OK: Double colon rules work independently of each other
whizprog.o:: foo.h
        $(CC) -c $(CFLAGS) whizprog.o
        @echo built for foo.h

whizprog.o:: bar.h
        $(CC) -c $(CFLAGS) whizprog.o
        @echo built for bar.h
```

No tab should precede any *targets*. (At the end of a dependency line, you can specify a command, preceded by a semicolon; however, commands are typically entered on their own lines, preceded by a tab.)

Targets of the form *library*(*member*) represent members of archive libraries, e.g., libguide.a(dontpanic.o). Furthermore, both targets and prerequisites may contain shell-style wildcards (e.g., *.c). make expands the wildcard and uses the resulting list for the targets or prerequisites.

Suffix rules

These specify that files ending with the first suffix can be prerequisites for files ending with the second suffix (assuming the root filenames are the same). Either of these formats can be used:

```
.suffix.suffix:
.suffix:
```

The second form means that the root filename depends on the filename with the corresponding suffix.

Pattern rules

Rules that use the % character define a pattern for matching targets and prerequisites. This is a powerful generalization of the original make's suffix rules. Many of GNU make's built-in rules are pattern rules. For example, this built-in rule is used to compile C programs into relocatable object files:

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

Each target listed in a pattern rule must contain only one % character. To match these rules, files must have at least one character in their names to match the %; a file named just .o would not match the above rule. The text that matches the % is called the *stem*, and the stem's value is substituted for the % in the prerequisite. (Thus, for example, prog.c becomes the prerequisite for prog.o.)

Conditional statements

Statements that evaluate conditions, and depending upon the result, include or exclude other statements from the contents of the makefile. More detail is given in the section "Conditional Input," later in this chapter.

Macro definitions

Macro definitions define variables: identifiers associated with blocks of text. Variable values can be created with either `=`, `:=`, or `define`, and appended to with `+=`. More detail is provided in the later section "Creating and Using Macros."

`include` *statements*

Similar to the C `#include` directive, there are three forms:

```
include file [file ...]
-include file [file ...]
sinclude file [file ...]
```

`make` processes the value of *file* for macro expansions before attempting to open the file. Furthermore, each *file* may be a shell-style wildcard pattern, in which case `make` expands it to produce a list of files to read.

The second and third forms have the same meaning. They indicate that `make` should try to include the named lines, but should continue without an error if a file could not be included. The `sinclude` version provides compatibility with other versions of `make`.

`vpath` *statements*

Similar to the `VPATH` variable, the `vpath` line has one of the following three forms:

`vpath pattern directory ...`	*Set directory list for pattern*
`vpath pattern`	*Clear list for pattern*
`vpath`	*Clear all lists*

Each *pattern* is similar to those for pattern rules, using `%` as a wildcard character. When attempting to find a prerequisite, `make` looks for a `vpath` rule that matches the prerequisite, and then searches in the directory list (separated by spaces or colons) for a matching file. Directories provided with `vpath` directives are searched *before* those provided by the `VPATH` variable.

Command lines

These lines are where you give the commands to actually rebuild those files that are out of date. Commands are grouped below the dependency line and are typed on lines that begin with a tab. If a command is preceded by a hyphen (–), `make` ignores any error returned. If a command is preceded by an at sign (`@`), the command line won't echo on the display (unless `make` is called with `-n`). Lines beginning with a plus (`+`) are always executed, even if `-n`, `-q`, or `-t` are used. This also applies to lines containing `$(MAKE)` or `${MAKE}`. Further advice on command lines is given later in this chapter.

Special Dependencies

GNU `make` has two special features for working with dependencies.

Library dependencies

A dependency of the form `-l`*`NAME`* causes `make` to search for a library file whose name is either `lib`*`NAME`*`.so` or `lib`*`NAME.a`* in the standard library directories. This is customizable with the `.LIBPATTERNS` variable; see the later section "Macros with Special Handling" for more information.

Order-only prerequisites

When a normal prerequisite of a target is out of date, two things happen. First, the prerequisite (and its prerequisites, recursively) are rebuilt as needed. This imposes an *ordering* on the building of targets and prerequisites. Second, after the prerequisites are updated, the target itself is rebuilt using the accompanying commands. Normally, both of these are what's desired.

Sometimes, you just wish to impose an ordering, such that the prerequisites are themselves updated, but the target is not rebuilt by running its rules. Such *order-only* prerequisites are specified in a dependency line by placing them to the right of a vertical bar or pipe symbol, `|`:

```
target: normal-dep1 normal-dep2 | order-dep1 order-dep2
        command
```

Dependency lines need not contain both. I.e., you do not have to provide regular dependencies if there are order-only dependencies as well; just place the | right after the colon.

Here is an annotated example of an order-only dependency:

```
$ cat Makefile
all: target                         First target is default, point to real target

prereq0:                            How to make prereq0
        @echo making prereq0
        touch prereq0

prereq1:                            How to make prereq1
        @echo making prereq1
        touch prereq1

prereq2: prereq0                    prereq2 depends on prereq0
        @echo making prereq2
        touch prereq2

target: prereq1 | prereq2           How to make target
        @echo making target
        touch target
```

The order of creation is shown in Figure 16-1.

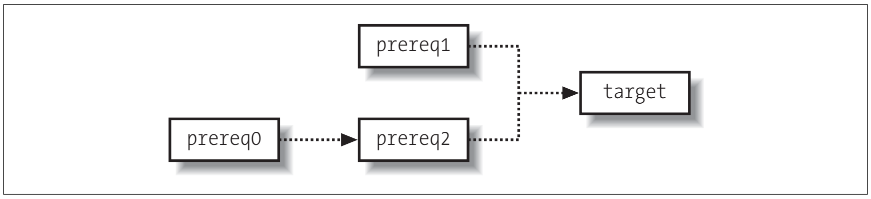


Figure 16-1. The order of creation

And here is the result of running `make`:

```
$ make
making prereq1
touch prereq1
```

```
making prereq0
touch prereq0
making prereq2
touch prereq2
making target
touch target
```

This is normal and as expected. Now, let's update one of the order-only prerequisites and rerun `make`:

```
$ touch prereq0
$ make
making prereq2
touch prereq2
```

Note that `target` was *not* rebuilt! Had the dependency on `prereq2` been a regular dependency, then `target` itself would also have been remade.

Conditional Input

Conditional statements allow you to include or exclude specific lines based on some condition. The condition can be that a macro is or is not defined, or that the value of a macro is or is not equal to a particular string. The equivalence/nonequivalence tests provide three different ways of quoting the values. Conditionals may have an optional "else" part; i.e., lines that are used when the condition is *not* true. The general form is as follows:

```
ifXXX test
    lines to include if true
[ else
    lines to include if false ]
endif
```

(The square brackets indicate optional parts of the construct; they are not to be entered literally.) Actual tests are as follows:

Condition	Meaning
`ifdef` *macroname*	True if *macroname* is a macro that has been given a value.
`ifndef` *macroname*	True if *macroname* is a macro that has *not* been given a value.

`ifeq (`*v1*`,`*v2*`)`
`ifeq '`*v1*`' '`*v2*`'`
`ifeq "`*v1*`" "`*v2*`"`
	True if values *v1* and *v2* are equal.

`ifneq (`*v1*`,`*v2*`)`
`ifneq '`*v1*`' '`*v2*`'`
`ifneq "`*v1*`" "`*v2*`"`
	True if values *v1* and *v2* are not equal.

For example:

```
whizprog.o: whizprog.c
ifeq($(ARCH),ENIAC)      # Serious retrocomputing in progess!
        $(CC) $(CFLAGS) $(ENIACFLAGS) -c $< -o $@
else
        $(CC) $(CFLAGS) -c $< -o $@
endif
```

Macros

This section summarizes creating and using macros, internal macros, macro modifiers, macros with special handling, and text manipulation with macros and functions.

Creating and Using Macros

Macros (often called *variables*) are like variables in a programming language. In `make`, they are most similar to variables in the shell language, having string values that can be assigned, referenced, and compared.

Defining macros

GNU `make` provides multiple ways to define macros. The different mechanisms affect how `make` treats the value being assigned. This in turn affects how the value is treated when the macro's value is retrieved, or *referenced*. GNU `make` defines two types of variables, called *recursively expanded variables* and *simply expanded variables*, respectively. The various macro assignment forms are as follows:

`name` `=` *`value`*

Create a recursively expanded variable. The value of *name* is the *verbatim text* on the right side of the `=`. If this value contains any references to other variable values, those values are retrieved and expanded when the original variable is referenced. For example:

```
bar = v1
foo = $(bar)              Value of bar retrieved when foo's value is referenced
...
x = $(foo)                x is assigned 'v1'
bar = v2
y = $(foo)                y is assigned 'v2'
```

`name` `:=` *`value`*

Create a simply expanded variable. The *value* is expanded completely, immediately at the time of the assignment. Any variable references in *value* are expanded then and there. For example:

```
bar = v1
foo := $(bar)             foo is assigned 'v1'
x = $(foo)                x is assigned 'v1'
bar = v2
y = $(foo)                y is still assigned 'v1'
```

A significant advantage of simply expanded variables is that they work like variables in most programming languages, allowing you to use their values in assignments to themselves:

```
x := $(x) other stuff
```

`name` `+=` *`value`*

Append *value* to the contents of variable *name*. If *name* was never defined, `+=` acts like `=`, creating a recursively defined variable. Otherwise, the result of `+=` depends upon the type of *name*. If *name* was defined with `=`, then *value* is appended literally to the contents of *name*. However, if *name* was defined with `:=`, then `make` completely expands *value* before appending it to the contents of *name*.

`name ?= value`
: Create recursively expanded variable *name* with value *value* only if *name* is not defined. Note that a variable that has been given an empty value is still considered to be defined.

```
define name
 ...
endef
```
Define a recursively expanded variable, similar to `=`. However, using `define`, you can give a macro a value that contains one or more newlines. This is not possible with the other assignment forms (`=`, `:=`, `+=`, `?=`).

Macro values

Macro values are retrieved by prefixing the macro name with a `$`. A plain `$` is enough for macros whose names are a single character, such as `$<` and `$@`. However, macro names of two or more characters must be enclosed in parentheses and preceded by a `$`. For example, `$(CC)`, `$(CPP)`, and so on.

Although it was not documented, the original V7 Unix version of `make` allowed the use of curly braces instead of parentheses: `${CC}`, `${RM}`, and so on.* All Unix versions and GNU `make` support this as well, and it is included in POSIX. This usage was particularly common in makefiles in the BSD distributions. There is no real reason to prefer one over the other, although long-time Unix programmers may prefer the parentheses form, since that is what was originally documented.

Exporting macros

By default, `make` exports variables to subprocesses only if those variables were already in the environment or if they were defined on the command line. Furthermore, only variables whose names contain just letters, digits, and underscores are exported, as many shells cannot handle environment variables with punctuation characters in their names. You can use the `export` directive to control exporting of specific variables, or all variables. The `unexport` directive indicates that a particular variable should *not* be exported; it cancels the effect of a previous `export` command. The command forms are as follows:

`export`
: By itself, the `export` directive causes `make` to export all alphanumerically named variables to the environment (where underscore counts as a letter too).

`export var`
: Export variable *var* to the environment. The variable will be exported even if its name contains nonalphanumeric characters.

* See the function `subst()` in *http://minnie.tuhs.org/UnixTree/V7/usr/src/cmd/make/misc.c.html*.

`export` *`var`* `=` *`value`*
`export` *`var`* `:=` *`value`*
`export` *`var`* `+=` *`value`*
`export` *`var`* `?` *`value`*

Perform the kind of assignment indicated by the given operator (as described earlier), and then export the variable to the environment.

`unexport` *`var`*

Do not export variable *var* to the environment. Cancels a previous `export` of *var* (for example, from a separate, included makefile).

Overriding command-line macros

Normally, when a macro is defined on the command line, the given value is used, and any value assigned to the macro within the makefile is ignored. Occasionally, you may wish to force a variable to have a certain value, or to append a value to a variable, no matter what value was given on the command line. This is the job of the `override` directive.

`override` *`var`* `=` *`value`*
`override` *`var`* `:=` *`value`*
`override` *`var`* `+=` *`value`*
`override` *`var`* `?` *`value`*
`override define` *`name`*

```
 ...
endef
```

Perform the kind of assignment indicated by the given operator (as described earlier), and then export the variable to the environment.

The example given in the GNU `make` documentation, *GNU Make: A Program for Directing Recompilation*, is forcing `CFLAGS` to always contain the `-g` option:

```
override CFLAGS += -g
```

Internal Macros

`$?`	The list of prerequisites that have been changed more recently than the current target. Can be used only in normal makefile entries—not suffix rules.
`$@`	The name of the current target, except in makefile entries for making libraries, where it becomes the library name. (For `libguide.a(dontpanic.o)`, `$@` is `libguide.a`). Can be used both in normal makefile entries and in suffix rules.
`$$@`	The name of the current target. Can be used only to the right of the colon in dependency lines. This is provided only for compatibility with System V `make`; its use is not recommended.
`$<`	The name of the current prerequisite that has been modified more recently than the current target.
`$*`	The name—without the suffix—of the current prerequisite that has been modified more recently than the current target. Should be used only in implicit rules or static pattern rules.
`$%`	The name of the corresponding `.o` file when the current target is a library module. (For `libguide.a(dontpanic.o)`, `$%` is `dontpanic.o`). Can be used both in normal makefile entries and in suffix rules.

`$^`	The list of prerequisites for the current target. For archive members, only the member name is listed. Even if a prerequisite appears multiple times in a dependency list for a target, it only appears once in the value of `$^`.
`$+`	Like `$^`, but prerequisites that appear multiple times in a dependency list for a target are repeated. This is most useful for libraries, since multiple dependencies upon a library can make sense and be useful.
`$$`	A literal `$` for use in rule command lines: for example, when referencing shell variables in the environment or within a loop.
`$|`	The order-only prerequisites for the current target.

Macro Modifiers

Macro modifiers may be applied to the built-in internal macros listed earlier, except for `$$`.

`D` The directory portion of any internal macro name. Valid uses are:

```
$(%D)    $(@D)
$(*D)    $$(@D)
$(<D)    $(^D)
$(?D)    @(+D)
```

`F` The file portion of any internal macro name. Valid uses are:

```
$(%F)    $(@F)
$(*F)    $$(@F)
$(<F)    $(^F)
$(?F)    @(+F)
```

Macros with Special Handling

`CURDIR`	The current working directory. Set by `make` but not used by it, for use in makefiles.
`.LIBPATTERNS`	Used for finding link library names as prerequisites of the form `-l`*`name`*. For each such prerequisite, `make` searches in the current directory, directories matching any `vpath` directives, directories named by the `VPATH` variable, `/lib`, `/usr/lib`, and *prefix*`/lib`, where *prefix* is the installation directory for GNU `make` (normally `/usr/local`). The default value of `.LIBPATTERNS` is `lib%.so lib%.a`. Thus `make` first searches for a shared library file, and then for a regular archive library.
`MAKE`	The full pathname used to invoke `make`. It is special because command lines containing the string `$(MAKE)` or `${MAKE}` are always executed, even when any of the `-n`, `-q`, or `-t` options are used.
`MAKECMDGOALS`	The targets given to `make` on the command line.
`MAKEFILE_LIST`	A list of makefiles read so far. The rightmost entry in the list is the name of the makefile currently being read.
`MAKEFILES`	Environment variable: `make` reads the whitespace-separated list of files named in it before reading any other makefiles.
`MAKEFLAGS`	Contains the flags inherited in the environment variable MAKEFLAGS, plus any command-line options. Used to pass the flags to subsequent invocations of `make`, usually via command lines in a makefile entry that contain `$(MAKE)`.
`MAKELEVEL`	The depth of recursion (sub-`make` invocation). Primarily for use in conditional statements so that a makefile can act in one way as the top-level makefile and in another way if invoked by another `make`.

`MAKOVERRIDES`	A list of the command-line variable definitions. `MAKEFLAGS` refers to this variable. By setting it to the empty string: `MAKEOVERRIDES =` You can pass down the command-line options to sub-`make`s but avoid passing down the variable assignments.
`MAKESHELL`	For MS-DOS only, the shell `make` should use for running commands.
`MFLAGS`	Similar to `MAKEFLAGS`, this variable is set for compatibility with other versions of `make`. It contains the same options as in `MAKEFLAGS`, but not the variable settings. It was designed for explicit use on command lines that invoke `make`. For example: `mylib:` `cd mylib && $(MAKE) $(MFLAGS)` The use of `MAKEFLAGS` is preferred.
`SHELL`	Sets the shell that interprets commands. If this macro isn't defined, the default is `/bin/sh`. On MS-DOS, if `SHELL` not set, the value of `COMSPEC` is used; see also the `MAKESHELL` variable, earlier in this list.
`SUFFIXES`	The default list of suffixes, before `make` reads and processes makefiles.
`.VARIABLES`	A list of all variables defined in all makefiles read up to the point that this variable is referenced.
`VPATH`	Specifies a list of directories to search for prerequisites when not found in the current directory. Directories in the list should be separated with spaces or colons.

Text Manipulation with Macros and Functions

Standard versions of `make` provide a limited text manipulation facility:

`$(`*`macro`*`:`*`s1`*`=`*`s2`*`)`
: Evaluates to the current definition of `$(`*`macro`*`)`, after substituting the string *s2* for every occurrence of *s1* that occurs either immediately before a blank or tab, or at the end of the macro definition.

GNU `make` supports this for compatibility with Unix `make` and the POSIX standard. However, GNU `make` goes *far* beyond simple text substitution, providing a host of *functions* for text manipulation. The following list provides a brief description of each function.

`$(addprefix` *`prefix, names ...`*`)`
: Generates a new list, created by prepending *prefix* to each of the *names*.

`$(addsuffix` *`suffix, names ...`*`)`
: Generates a new list, created by appending *suffix* to each of the *names*.

`$(basename` *`names ...`*`)`
: Returns a list of the *basename* of each of the *names*. The basename is the text up to but not including the final period.

`$(call` *`var, param, ...`*`)`
: The `call` function allows you to treat the value of a variable as a procedure. *var* is the *name* of a variable, not a variable reference. The *param*s are assigned to temporary variables that may be referenced as `$(1)`, `$(2)`, and so on. `$(0)` will be the name of the variable. The value of *var* should reference the temporary values. The result of `call` is the result of evaluating *var* in this way. If *var* names a built-in function, that function is always called, even if a

`make` variable of the same name exists. Finally, `call` may be used recursively; each invocation gets its own `$(1)`, `$(2)`, and so on.

`$(dir` *`names …`*`)`
: Returns a list of the *directory part* of each of the *names*. The directory part is all text, up to and including the final `/` character. If there is no `/`, the two characters `./` are used.

`$(error` *`text …`*`)`
: Causes `make` to produce a fatal error message consisting of *text*.

`$(filter` *`pattern …, text`*`)`
: Chooses the words in *text* that match any *pattern*. Patterns are written using `%`, as for the `patsubst` function.

`$(filter-out` *`pattern …, text`*`)`
: Like `filter`, but selects the words that do *not* match the patterns.

`$(findstring` *`find, text`*`)`
: Searches *text* for an instance of *find*. If found, the result is *find*; otherwise, it's the empty string.

`$(firstword` *`names …`*`)`
: Returns the first word in *names*.

`$(foreach` *`var, words, text`*`)`
: This function is similar to the `for` loop in the shell. It expands *var* and *words*, first. The result of expanding *var* names a macro. `make` then loops, setting *var* to each word in *words*, and then evaluating *text*. The result is the concatenation of all the iterations. The *text* should contain a reference to the variable for this to work correctly.

  If *var* is defined before the `foreach` is evaluated, it maintains the same value it had after the evaluation. If it was undefined before the `foreach`, it remains undefined afterwords. In effect, `foreach` creates a temporary, private variable named *var*.

`$(if` *`condition, then-text`*`[,` *`else-text`*`])`
: The *condition* is evaluated. If, after removing leading and trailing whitespace, the result is not empty, the condition is considered to be true, and the result of `if` is the expansion of the *then-text*. Otherwise, the condition is considered to be false, and the result is the expansion of *else-text*, if any. If there's no *else-text*, then a false condition produces the empty string. Only one or the other of *then-text* and *else-text* is evaluated.

`$(join` *`list1, list2`*`)`
: Produces a new list where the first element is the concatenation of the first elements in *list1* and *list2*, the second element is the concatenation of the second elements in *list1* and *list2*, and so on.

`$(notdir` *`names …`*`)`
: Returns a list of the nondirectory part of each of the *names*. The nondirectory part is all the text after the final `/`, if any. If not, it's the entire *name*.

`$(origin` *`variable`*`)`

Returns a string describing the origin of *variable*. Here, *variable* is a variable name (`foo`), not a variable reference (`$(foo)`). Possible return values are one of the following:

`automatic`	The variable is an automatic variable for use in the commands of rules, such as `$*` and `$@`.
`command line`	The variable was defined on the command line.
`default`	The variable is one of those defined by `make`'s built-in rules, such as `CC`.
`environment`	The variable was defined in the environment, and `-e` was *not* used.
`environment override`	The variable was defined in the environment, and `-e` *was* used.
`file`	The variable was defined in a makefile.
`override`	The variable was defined with an `override` command. See the earlier section "Overriding command-line macros."
`undefined`	The variable was never given a value.

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

Replaces words in *text* that match *pattern* with *replacement*. The *pattern* should use a `%` as a wildcard character. In *replacement*, a `%` acts as the placeholder for the text that matched the `%` in *pattern*. This is a general form of string substitution. For example, the traditional `OBJS = $(SRCS:.c=.o)` could instead be written `OBJS = $(patsubst %.c, %.o, $(SRCS))`.

`$(shell` *`command`*`)`

Runs the shell command *command* and returns the output. `make` converts newlines in the output into spaces and removes trailing newlines. This is similar to \`...\` in the shell.

`$(sort` *`list`*`)`

Returns a sorted copy of the words in *list*, with duplicates removed. Each word is separated from the next by a single space.

`$(subst` *`from, to, text`*`)`

Replaces every instance of *from* in *text* with *to*.

`$(suffix` *`names`* `...)`

Returns a list of the suffixes of each *name*. The suffix is the final period and any following text. Returns an empty string for a *name* without a period.

`$(strip` *`string`*`)`

Removes leading and trailing whitespace from *string* and converts internal runs of whitespace into single spaces. This is especially useful in conjunction with conditionals.

`$(warning` *`text`* `...)`

Causes `make` to produce a warning message consisting of *text*.

`$(wildcard` *`pattern`* `...)`

Creates a space-separated list of filenames that match the shell pattern *pattern*. (Note! Not a `make`-style `%` pattern.)

`$(word` *`n, text`*`)`

Returns the *n*th word of *text*, counting from one.

`$(wordlist` *`start, end, text`*`)`
: Creates a new list consisting of the words *start* to *end* in *text*. Counting starts at one.

`$(words` *`text`*`)`
: Returns the number of words in *text*.

Special Target Names

`.DEFAULT:`	Commands associated with this target are executed if `make` can't find any makefile entries or suffix rules with which to build a requested target.
`.DELETE_ON_ERROR:`	If this target appears in a makefile, then for any target that `make` is rebuilding, if its command(s) exit with a nonzero status, `make` deletes the target.
`.EXPORT_ALL_VARIABLES:`	The mere existence of this target causes `make` to export all variables to child processes.
`.IGNORE:`	With prerequisites, ignore problems just for those files. For historical compatibility, with no prerequisites, ignore error returns from all commands. This is the same as the `-i` option.
`.INTERMEDIATE:`	Prerequisites for this target are treated as intermediate files, even if they are mentioned explicitly in other rules. (An intermediate file is one that needs to be built "along the way" to the real target. For example, making a `.c` file from a `.y` file, in order to create a `.o` object file. The `.c` file is an intermediate file.) This prevents them from being re-created, unless one of their prerequisites is out of date.
`.LOW_RESOLUTION_TIME:`	`make` notes that prerequisites for this target are updated by commands that only create low resolution timestamps (one second granularity). For such targets, if their modification time starts at the same second as the modification time of a prerequisite, `make` does not try to compare the sub-second time values, and does not treat the file as being out of date.
`.NOTPARALLEL:`	Prerequisites for this target are ignored. Its existence in a makefile overrides any `-j` option, forcing all commands to run serially. Recursive `make` invocations may still run jobs in parallel, unless their makefiles also contain this target.
`.POSIX:`	When this target exists, changing the `MAKEOVERRIDES` variable does *not* affect the `MAKEFLAGS` variable. (This is a rather specialized case.) This target also disables the special treatment of `$$@`, `$$(@D)`, and `$$(@F)`.
`.PHONY:`	Prerequisites for this target are marked as "phony." I.e., `make` always executes their rules, even if a file by the same name exists.
`.PRECIOUS:`	Files you specify for this target are not removed when you send a signal (such as interrupt) that aborts `make`, or when a command line in your makefile returns an error.
`.SECONDARY:`	Prerequisites of this target are treated like intermediate files, except that they are never automatically removed. With no prerequisites, all targets are treated as secondary.
`.SILENT:`	When given prerequisites, `make` will not print the commands for those prerequisites when they are rebuilt. Otherwise, for historical compatibility, when this target has no prerequisites, `make` executes all commands silently, which is the same as the `-s` option.
`.SUFFIXES:`	Suffixes associated with this target are meaningful in suffix rules. If no suffixes are listed, the existing suffix rules are effectively "turned off."

Writing Command Lines

Writing good, portable makefile files is a bit of an art. Skill comes with practice and experience. Here are some tips to get you started:

- Depending upon your locale, naming your file `Makefile` instead of `makefile` can cause it to be listed first with `ls`. This makes it easier to find in a directory with many files.
- Remember that command lines must start with a leading tab character. You cannot just indent the line with spaces, even eight spaces. If you use spaces, `make` exits with an unhelpful message about a "missing separator."
- Remember that `$` is special to `make`. To get a literal `$` into your command lines, use `$$`. This is particularly important if you want to access an environment variable that isn't a `make` macro. Also, if you wish to use the shell's `$$` for the current process ID, you have to type it as `$$$$`.
- Write multiline shell statements, such as shell conditionals and loops, with trailing semicolons and a trailing backslash:

  ```
  if [ -f specfile ] ; then \
  ... ; \
  else \
  ... ; \
  fi
  ```

  Note that the shell keywords `then` and `else` don't need the semicolon. (What happens is that `make` passes the backslashes and the newlines to the shell. The escaped newlines are not syntactically important, so the semicolons are needed to separate the different parts of the command. This can be confusing. If you use a semicolon where you would normally put a newline in a shell script, things should work correctly.)
- Remember that each line is run in a separate shell. This means that commands that change the shell's environment (such as `cd`) are ineffective across multiple lines. The correct way to write such commands is to keep the commands on the same line, separated with a semicolon. In the particular case of `cd`, separate the commands with `&&` in case the subdirectory doesn't exist or can't be changed to:

  ```
  cd subdir && $(MAKE)
  ...
  PATH=special-path-value ; export PATH ; $(MAKE)
  ```
- For guaranteed portability, always set `SHELL` to `/bin/sh`. Some versions of `make` use whatever value is in the environment for `SHELL`, unless it is explicitly set in the makefile.
- Use macros for standard commands. `make` already helps out with this, providing macros such as `$(CC)`, `$(YACC)`, and so on.
- When removing files, start your command line with `-$(RM)` instead of `$(RM)`. (The – causes `make` to ignore the exit status of the command.) This way, if the file you were trying to remove doesn't exist, and `rm` exits with an error, `make` can keep going.

- When running subsidiary invocations of `make`, typically in subdirectories of your main program tree, always use `$(MAKE)`, and not `make`. Lines that contain `$(MAKE)` are always executed, even if `-n` has been provided, allowing you to test out a whole hierarchy of makefiles. This does not happen for lines that invoke `make` directly.
- Often, it is convenient to organize a large software project into subprojects, with each one having a subdirectory. The top-level makefile then just invokes `make` in each subdirectory. Here's the way to do it:

```
SUBDIRS = proj1 proj2 proj3
...
projects: $(SUBDIRS)
        for i in $(SUBDIRS); \
        do \
                echo ====== Making in $$i ; \
                ( cd $$i && $(MAKE) $(MAKEFLAGS) $@ ) ; \
        done
```