make *Utility* 4

This chapter describes the *make* utility, which includes:

- Hidden dependency checking
- Command dependency checking
- Pattern-matching rules
- Automatic retrieval of SCCS files

This version of the make utility runs successfully with makefiles written for previous versions of make. Makefiles that rely on enhancements may not be compatible with other versions of this utility (see Appendix A, “System V make” for more information on previous versions of make). Refer to “make Enhancements Summary” on page 216 for a complete summary of enhancements and compatibility issues.

make streamlines the process of generating and maintaining object files and executable programs. It helps you to compile programs consistently and eliminates unnecessary recompilation of modules that are unaffected by source code changes.

make provides features that simplify compilations. You can also use it to automate any complicated or repetitive task that is not interactive. You can use make to update and maintain object libraries, to run test suites, and to install files onto a filesystem or tape. In conjunction with SCCS, you can use make to ensure that a large software project is built from the desired versions in an entire hierarchy of source files.

make reads a file that you create, called a *makefile*, which contains information about what files to build and how to build them. Once you write and test the makefile, you can forget about the processing details; *make* takes care of them.

Dependency Checking: make *vs. Shell Scripts*

While it is possible to use a shell script to assure consistency in trivial cases, scripts to build software projects are often inadequate. On the one hand, you don't want to wait for a simpleminded script to compile every single program or object module when only one of them has changed. On the other hand, having to edit the script for each iteration can defeat the goal of consistency. Although it is possible to write a script of sufficient complexity to recompile only those modules that require it, make does this job better.

make allows you to write a simple, structured listing of what to build and how to build it. It uses the mechanism of *dependency checking* to compare each module with the source or intermediate files it derives from. make only rebuilds a module if one or more of these prerequisite files, called *dependency files*, has changed since the module was last built.

To determine whether a derived file is out of date with respect to its sources, make compares the modification time of the (existing) module with that of its dependency file. If the module is missing, or if it is older than the dependency file, make considers it to be out of date, and issues the commands necessary to rebuild it. A module can be treated as out of date if the commands used to build it have changed.

Because make does a complete dependency scan, changes to a source file are consistently propagated through any number of intermediate files or processing steps. This lets you specify a hierarchy of steps in a top to bottom fashion.

You can think of a makefile as a recipe. make reads the recipe, decides which steps need to be performed, and executes only those steps that are required to produce the finished module. Each file to build, or step to perform, is called a *target*. The makefile entry for a target contains its name, a list of targets on which it depends, and a list of commands for building it.

The list of commands is called a *rule*. make treats dependencies as prerequisite targets, and updates them (if necessary) before processing its current target. The rule for a target need not always produce a file, but if it does, the file for which the target is named is referred to as the *target file*. Each file from which a target is derived (for example, that the target depends on) is called a *dependency file*.

If the rule for a target produces no file by that name, `make` performs the rule and considers the target to be up-to-date for the remainder of the run.

`make` assumes that only *it* will `make` changes to files being processed during the current run. If a source file is changed by another process while `make` is running, the files it produces may be in an inconsistent state.

Writing a Simple Makefile

The basic format for a makefile target entry is shown in the following figure:

```
target  .   .   .   : [ dependency . . . ]
        [  command ]
        .   .   .
```

Figure 4-1 Makefile Target Entry Format

In the first line, the list of target names is terminated by a colon. This, in turn, is followed by the dependency list if there is one. If several targets are listed, this indicates that each such target is to be built independently using the rule supplied.

Subsequent lines that start with a TAB are taken as the command lines that comprise the target rule. A common error is to use SPACE characters instead of the leading TAB

Lines that start with a # are treated as comments up until the next (unescaped) NEWLINE and do not terminate the target entry. The target entry is terminated by the next non-empty line that begins with a character other than TAB or #, or by the end of the file.

A trivial makefile might consist of just one target shown in the following figure:

```
test:
	ls test
	touch test
```

Figure 4-2 A Trivial Makefile

When you run make with no arguments, it searches first for a file named makefile, or if there is no file by that name, Makefile. If either of these files is under SCCS control, make checks the makefile against its history file. If it is out of date, make extracts the latest version.

If make finds a makefile, it begins the dependency check with the first target entry in that file. Otherwise you must list the targets to build as arguments on the command line. make displays each command it runs while building its targets.

```
$ make
ls test
test not found
touch test
$ ls test
test
```

Because the file test was not present (and therefore out of date), make performed the rule in its target entry. If you run make a second time, it issues a message indicating that the target is now up to date and skips the rule:

```
$ make
'test' is up to date.
```

make invokes a Bourne shell to process a command line if that line contains any shell metacharacters, such as a semicolon (;), redirection symbols (<, >, >>, |), substitution symbols (*, ?, ,[] $, =), or quotes, escapes or comments (", ', ‘, \, #, etc.:), If a shell isn't required to parse the command line, make exec()'s the command directly.

Line breaks within a rule are significant in that each command line is performed by a separate process or shell.

This means that a rule such as:

```
test:
    cd /tmp
    pwd
```

behaves differently than you might expect, as shown below.

```
$ make test
cd /tmp
pwd
/usr/tutorial/waite/arcana/minor/pentangles
```

You can use semicolons to specify a sequence of commands to perform in a single shell invocation:

```
test:
    cd /tmp ; pwd
```

Or, you can continue the input line onto the next line in the makefile by escaping the NEWLINE with a backslash (\). The escaped NEWLINE is treated as white space by `make`.

The backslash must be the last character on the line. The semicolon is required by the shell.

```
test:
    cd /tmp ; \
    pwd
```

Basic Use of Implicit Rules

When no rule is given for a specified target, `make` attempts to use an *implicit rule* to build it. When `make` finds a rule for the class of files the target belongs to, it applies the rule listed in the implicit rule target entry.

In addition to any makefile(s) that you supply, `make` reads in the default makefile, `/usr/share/lib/make/make.rules`, which contains the target entries for a number of implicit rules, along with other information.[1]

There are two types of implicit rules. *Suffix* rules specify a set of commands for building a file with one suffix from another file with the same base name but a different suffix. *Pattern-matching* rules select a rule based on a target and dependency that match respective wild-card patterns. The implicit rules provided by default are suffix rules.

In some cases, the use of suffix rules can eliminate the need for writing a makefile entirely. For instance, to build an object file named `functions.o` from a single C source file named `functions.c`, you could use the command:

```
$ make functions.o
cc -c functions.c -o functions.o
```

1. Implicit rules were hand-coded in earlier versions of `make`.

This would work equally well for building the object file `nonesuch.o` from the source file `nonesuch.c`.

To build an executable file named `functions` (with a null suffix) from `functions.c`, you need only type the command:

```
$ make functions
cc -o functions functions.c
```

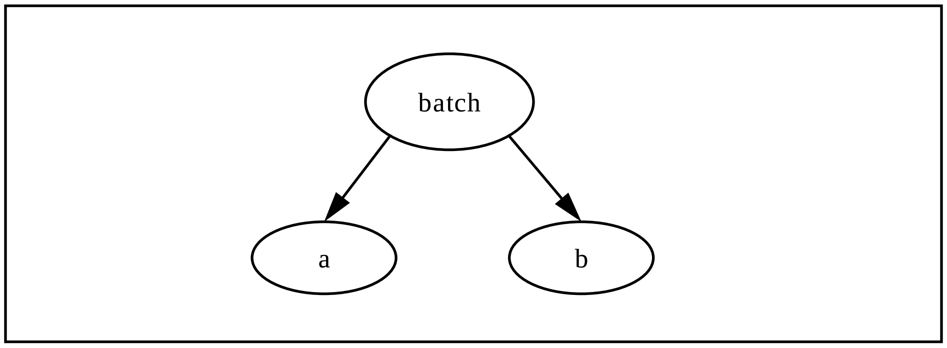
The rule for building a `.o` file from a `.c` file is called the `.c.o` (pronounced "dot-see-dot-oh") suffix rule. The rule for building an executable program from a `.c` file is called the `.c` rule. The complete set of default suffix rules is listed in Table 4-2 on page 174.

Processing Dependencies

Once `make` begins, it processes targets as it encounters them in its depth-first dependency scan. For example, with the following `makefile`:

```
batch: a b
    touch batch
b:
    touch b
a:
    touch a
c:
    echo "you won't see me"
```

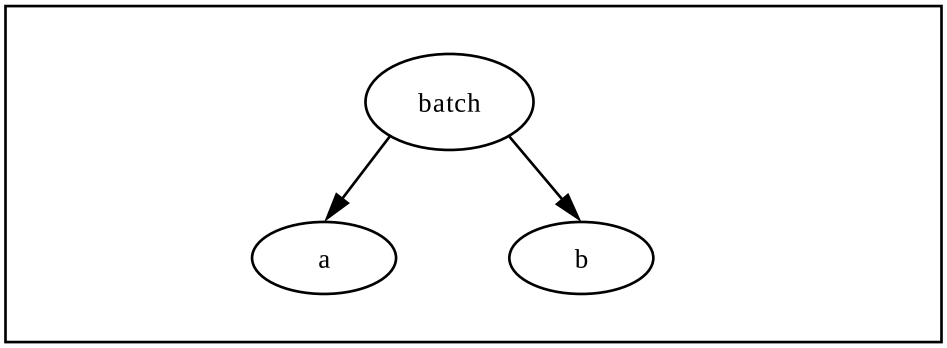
make starts with the target batch. Since batch has some dependencies that haven’t been checked, namely a and b, make defers batch until after it has checked them against any dependencies they might have.



Since a has no dependencies, make processes it; if the file is not present, make performs the rule in its target entry.

```
$ make
touch a
...
```

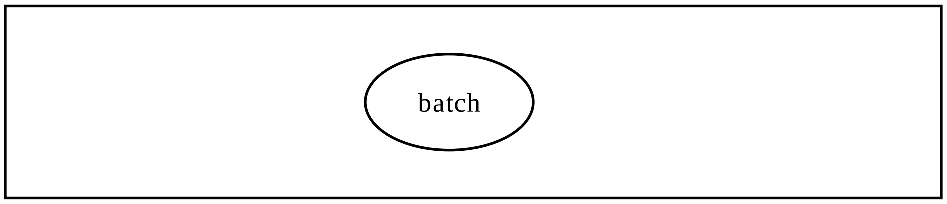
Next, make works its way back up to the parent target batch. Since there is still an unchecked dependency b, make descends to b and checks it.



b also has no dependencies, so make performs its rule:

```
...
touch b
...
```

Finally, now that all of the dependencies for `batch` have been checked and built (if needed), `make` checks `batch`.



Since it rebuilt at least one of the dependencies for `batch`, `make` assumes that `batch` is out of date and rebuilds it; if `a` or `b` had not been built in the current `make` run, but were present in the directory and newer than `batch`, `make`'s time stamp comparison would also result in `batch` being rebuilt:

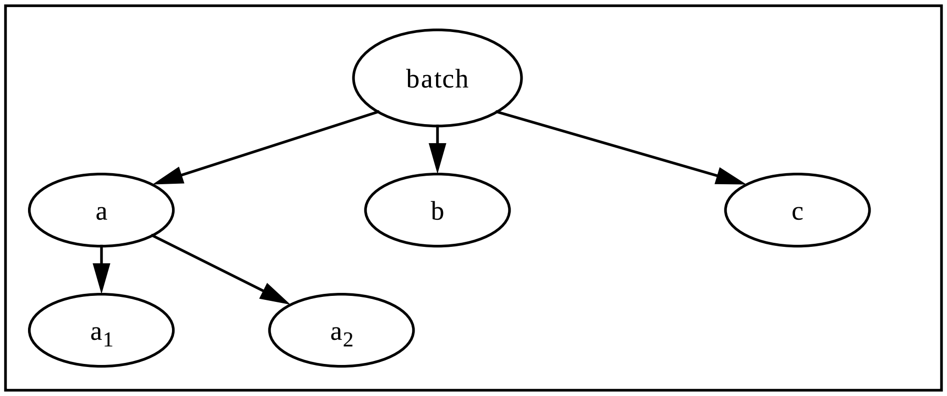
```
...
touch batch
```

Target entries that aren't encountered in a dependency scan are not processed. Although there is a target entry for `c` in the makefile, `make` does not encounter it while performing the dependency scan for `batch`, so its rule is not performed. You can select an alternate starting target like `c` by entering it as an argument to the `make` command.

In the next example, the `batch` target produces no file. Instead, it is used as a label to group a set of targets.

```
batch: a b c
a: a1 a2
    touch a
b:
    touch b
c:
    touch c
a1:
    touch a1
a2:
    touch a2
```

In this case, the targets are checked and processed, as shown in the following diagram:



Essentially, make attempts to:

1. Check batch for dependencies and notes that there are three, and so defers it.
2. Check a, the first dependency, and notes that it has two dependencies of its own. Continuing in like fashion, make:
   a. Checks a1, and if necessary, rebuilds it.
   b. Checks a2, and if necessary, rebuilds it.
3. Determines whether to build a.
4. Checks b and rebuilds it if need be.
5. Checks and rebuilds c if needed.
6. After traversing its dependency tree, make checks and processes the topmost target, batch. If batch contained a rule, make would perform that rule. Since batch has no rule, make performs no action, but notes that batch has been rebuilt; any targets depending on batch would also be rebuilt.

Null Rules

If a target entry contains no rule, make attempts to select an implicit rule to build it. If make cannot find an appropriate implicit rule and there is no SCCS history from which to retrieve it, make concludes that the target has no corresponding file, and regards the missing rule as a null rule. With this makefile:

You can use a dependency with a null rule to force the target rule to be executed. The conventional name for such a dependency is FORCE.

```
haste: FORCE
    echo "haste makes waste"
FORCE:
```

make performs the rule for making haste, even if a file by that name is up to date:

```
$ touch haste
$ make haste
echo "haste makes waste"
haste makes waste
```

Special Targets

make has several built-in *special targets* that perform special functions. For example, the .PRECIOUS special target directs make to preserve library files when make is interrupted.

Special targets:

- begin with a period (.)
- have no dependencies
- can appear anywhere in a makefile

Table 4-1 on page 153 includes a list of special targets.

Unknown Targets

If a target is named either on the command line or in a dependency list, and it

- is not a file present in the working directory
- has no target or dependency entry
- does not belong to a class of files for which an implicit rule is defined

- has no SCCS history file, and
- there is no rule specified for the `.DEFAULT` special target

`make` stops processing and issues an error message.[1]

```
$ make believe
make: Fatal error: Don’t know how to make target ‘believe’.
```

Duplicate Targets

Targets may appear more than once in a makefile. For example,

```
foo:   dep_1
foo:   dep_2
foo:
    touch foo
```

is the same as

```
foo:   dep_1 dep_2
    touch foo
```

However, many people feel that it’s preferable to have a target appear only once, for ease of reading.

Reserved `make` *Words*

The words in the following table are reserved by `make`:

Table 4-1 Reserved `make` Words

.BUILT_LAST_MAKE_RUN	.DEFAULT	.DERIVED_SRC
.DONE	.IGNORE	.INIT
.KEEP_STATE	.MAKE_VERSION	.NO_PARALLEL
.PRECIOUS	.RECURSIVE	.SCCS_GET
.SILENT	.SUFFIXES	.WAIT

1. However, if the `-k` option is in effect, `make` will continue with other targets that do not depend on the one in which the error occurred.

Table 4-1 Reserved `make` Words

FORCE	HOST_ARCH	HOST_MACH
KEEP_STATE	MAKE	MAKEFLAGS
MFLAGS	TARGET_ARCH	TARGET_MACH
VERSION_1.0	VIRTUAL_ROOT	VPATH

Running Commands Silently

You can inhibit the display of a command line within a rule by inserting an `@` as the first character on that line. For example, the following target:

```
quiet:
    @echo you only see me once
```

produces:

```
$ make quiet
you only see me once
```

If you want to inhibit the display of commands during a particular `make` run, you can use the `–s` option. If you want to inhibit the display of all command lines in every run, add the special target `.SILENT` to your makefile.

```
.SILENT:
quiet:
    echo you only see me once
```

Special-function targets begin with a dot (`.`). Target names that begin with a dot are never used as the starting target, unless specifically requested as an argument on the command line. `make` normally issues an error message and stops when a command returns a nonzero exit code. For example, if you have the target:

```
rmxyz:
    rm xyz
```

and there is no file named xyz, make halts after rm returns its exit status.

```
$ ls xyz
xyz not found
$ make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1
make: Fatal error: Command failed for target ‘rmxyz’
```

If - and @ are the first two such characters, both take effect.

To continue processing regardless of the command exit code, use a dash character (-) as the first non-TAB character:

```
rmxyz:
    -rm xyz
```

In this case you get a warning message indicating the exit code make received:

```
$ make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1 (ignored)
```

Unless you are testing a makefile, it is usually a bad idea to ignore non-zero error codes on a global basis.

Although it is generally ill-advised to do so, you can have make ignore error codes entirely with the -i option. You can also have make ignore exit codes when processing a given makefile, by including the .IGNORE special target, though this too should be avoided.

If you are processing a list of targets, and you want make to continue with the next target on the list rather than stopping entirely after encountering a non-zero return code, use the −k option.

Automatic Retrieval of SCCS Files

When source files are named in the dependency list, make treats them just like any other target. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile.

When a target has no dependencies, but is present in the directory, `make` assumes that file is up to date. If, however, a source file is under SCCS control, `make` does some additional checking to assure that the source file is up to date. If the file is missing, or if the history file is newer, `make` automatically issues the following command to retrieve the most recent version:[1]

```
sccs get -s filename -Gfilename
```

However, if the source file is writable by anyone, `make` does not retrieve a new version.

```
$ ls SCCS/*
SCCS/s.functions.c
$ rm -f functions.c
$ make functions
sccs get -s functions.c -Gfunctions.c
cc -o functions functions.c
```

`make` only checks the time stamp of the retrieved version against the time stamp of the history file. It does *not* check to see if the version present in the directory is the most recently checked-in version. So, if someone has done a get by date (`sccs get -c`), `make` would not discover this fact, and you might unwittingly build an older version of the program or object file. To be absolutely sure that you are compiling the latest version, you can precede `make` with an `sccs get SCCS`‘ or an `sccs clean` command.

Suppressing SCCS Retrieval

The command for retrieving SCCS files is specified in the rule for the `.SCCS_GET` special target in the default makefile. To suppress automatic retrieval, simply add an entry for this target with an empty rule to your makefile:

```
# Suppress sccs retrieval.
    .SCCS_GET:
```

1. With other versions of `make`, automatic `sccs`retrieval was a feature only of certain implicit rules. Also, unlike earlier versions, `make` only looks for history (s.)files in the sccs directory; history files in the current working directory are ignored.

Passing Parameters: Simple `make` *Macros*

The `make` macro substitution comes in handy when you want to pass parameters to command lines within a makefile. Suppose that you want to compile an optimized version of the program `program` using `cc`'s `-O` option. You can lend this sort of flexibility to your makefile by adding a *macro reference*, such as the following example, to the target for `functions`:

```
functions: functions.c
    cc $(CFLAGS) -o functions functions.c
```

The macro reference acts as a placeholder for a value that you define, either in the makefile itself, or as an argument to the `make` command. If you then supply `make` with a *definition* for the `CFLAGS` macro, `make` replaces its references with the value you have defined.

There is a reference to the `CFLAGS` macro in both the `.c` and the `.c.o` implicit rules.
The command-line definition must be a single argument, hence the quotes in this example.

```
$ rm functions
$ make functions "CFLAGS= -O"
cc -O -o functions functions.c
```

If a macro is undefined, `make` expands its references to an empty string.

You can also include macro definitions in the makefile itself. A typical use is to set `CFLAGS` to `-O`, so that `make` produces optimized object code by default:

```
CFLAGS= -O
functions: functions.c
    cc $(CFLAGS) -o functions functions.c
```

A macro definition supplied as a command line argument to `make` overrides other definitions in the makefile.[1] For instance, to compile `functions` for debugging with `dbx` or `dbxtool`, you can define the value of `CFLAGS` to be `-g` on the command line:

```
$ rm functions
$ make CFLAGS=-g
cc -g -o functions functions.c
```

1. Conditionally defined macros are an exception to this. Refer to "Conditional Macro Definitions" on page 187 for details.

To compile a profiling variant for use with `gprof`, supply both `-O` and `-pg` in the value for `CFLAGS`.

A macro reference must include parentheses when the name of the macro is longer than one character. If the macro name is only one character, the parentheses can be omitted. You can use curly braces, `{` and `}`, instead of parentheses. For example, '`$X`', '`$(X)`', and '`${X}`' are equivalent.

`.KEEP_STATE` *and Command Dependency Checking*

In addition to the normal dependency checking, you can use the special target `.KEEP_STATE` to activate *command dependency* checking. When activated, `make` not only checks each target file against its dependency files, it compares each command line in the rule with those it ran the last time the target was built. This information is stored in the `.make.state` file in the current directory (see page 159).

With the makefile:

```
CFLAGS= -O
.KEEP_STATE:

functions: functions.c
    cc -o functions functions.c
```

the following commands work as shown:

```
$ make
cc -O -o functions functions.c
$ make CFLAGS=-g
cc -g -o functions functions.c
$ make "CFLAGS= -O -pg"
cc -O -pg -o functions functions.c
```

This ensures you that `make` compiles a program with the options you want, even if a different variant is present and otherwise up to date.

The first `make` run with `.KEEP_STATE` in effect recompiles all targets in order to gather and record the necessary information.

The `KEEP_STATE` variable, when imported from the environment, has the same effect as the `.KEEP_STATE` target.

Suppressing or Forcing Command Dependency Checking for Selected Lines

To suppress command dependency checking for a given command line, insert a question mark as the first character after the TAB.

Command dependency checking is automatically suppressed for lines containing the dynamic macro `$?`. This macro stands for the list of dependencies that are newer than the current target, and can be expected to differ between any two `make` runs.[1]

To force `make` to perform command dependency checking on a line containing this macro, prefix the command line with a `!` character (following the TAB).

The State File

When `.KEEP_STATE` is in effect, `make` writes out a state file named `.make.state`, in the current directory. This file lists all targets that have ever been processed while `.KEEP_STATE` has been in effect, along with the rules to build them, in makefile format. In order to assure that this state file is maintained consistently, once you have added `.KEEP_STATE` to a makefile, it is recommended that you leave it in effect.[2]

`.KEEP_STATE` *and Hidden Dependencies*

When a C source file contains `#include` directives for interpolating headers, the target depends just as much on those headers as it does on the sources that include them. Because such headers may not be listed explicitly as sources in the compilation command line, they are called *hidden dependencies*. When `.KEEP_STATE` is in effect, `make` receives a report from the various compilers and compilation preprocessors indicating which hidden dependency files were interpolated for each target.

1. See "Implicit Rules and Dynamic Macros" on page 168 for more information.

2. Since this target is ignored in earlier versions of `make`, it does not introduce any compatibility problems. Other versions simply treat it as a superfluous target that no targets depend on, with an empty rule and no dependencies of its own. Since it starts with a dot, it is not used as the starting target.

It adds this information to the dependency list in the state file. In subsequent runs, these additional dependencies are processed just like regular dependencies. This feature automatically maintains the hidden dependency list for each target; it insures that the dependency list for each target is always accurate and up to date. It also eliminates the need for the complicated schemes found in some earlier makefiles to generate complete dependency lists.

A slight inconvenience can arise the first time `make` processes a target with hidden dependencies, because there is as yet no record of them in the state file. If a header is missing, and `make` has no record of it, `make` won't know that it needs to retrieve it from SCCS before compiling the target.

Even though there is an SCCS history file, the current version won't be retrieved because it doesn't yet appear in a dependency list or the state file. When the C preprocessor attempts to interpolate the header, it won't find it; the compilation fails.

Supposing that a `#include` directive for interpolating the header `hidden.h` is added to `functions.c`, and that the file `hidden.h` is somehow removed before the subsequent `make` run. The results would be:

```
$ rm -f hidden.h
$ make functions
cc -O -o functions functions.c
functions.c: 2: Can't find include file hidden.h
make: Fatal error: Command failed for target 'functions'
```

A simple workaround might be to make sure that the new header is extant before you run `make`. Or, if the compilation should fail (and assuming the header is under SCCS), you could manually retrieve it from SCCS:

```
$ sccs get hidden.h
1.1
10 lines
$ make functions
cc -O -o functions functions.c
```

In all future cases, should the header turn up missing, `make` will know to build or retrieve it for you because it will be listed in the state file as a hidden dependency.

Note that with hidden dependency checking, the `$?` macro includes the names of hidden dependency files. This may cause unexpected behavior in existing makefiles that rely on `$?`.

`.INIT` *and Hidden Dependencies*

The problem with both of these approaches is that the first `make` in the local directory may fail due to a random condition in some other (include) directory. This might entail forcing someone to monitor a (first) build. To avoid this, you can use the `.INIT` target to retrieve known hidden dependencies files from SCCS. `.INIT` is a special target that, along with its dependencies, is built at the start of the `make` run. To be sure that `hidden.h` is present, you could add the following line to your makefile

```
.INIT:  hidden.h
```

Displaying Information About a `make` *Run*

Running `make` with the `-n` option displays the commands `make` is to perform, without executing them. This comes in handy when verifying that the macros in a makefile are expanded as expected. With the following makefile:

```
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
    $(LINK.c) -o functions main.o data.o
```

`make -n` displays:

```
$ make -n
cc -O -c main.c
cc -O -c data.c
cc -O -o functions main.o data.o
```

Note – There is an exception however. `make` executes any command line containing a reference to the `MAKE` macro (i.e., `$(MAKE)` or `${MAKE}`), regardless of `-n`. It would be a very bad idea to include a line such as the following in your makefile: `$(MAKE) ; rm -f *`

Setting an environment variable named MAKEFLAGS can lead to complications, since `make` adds its value to the list of options. To prevent puzzling surprises, avoid setting this variable.

`make` has some other options that you can use to keep abreast of what it's doing and why:

`-d`
Displays the criteria by which `make` determines that a target is be out-of-date. Unlike `-n`, it *does* process targets, as shown in the following example. This options also displays the value imported from the environment (null by default) for the `MAKEFLAGS` macro, which is described in detail in a later section.

```
$ make -d
MAKEFLAGS value:
    Building main.o using suffix rule for .c.o because it is out
of date relative to main.c
cc -O -c main.c
    Building functions because it is out of date relative to
main.o
    Building data.o using suffix rule for .c.o because it is out
of date relative to data.c
cc -O -c data.c
    Building functions because it is out of date relative to
data.o
cc -O -o functions main.o data.o
```

`-dd`
This option displays all dependencies `make` checks, including any hidden dependencies, in vast detail.

`-D`
Displays the text of the makefile as it is read.

`-DD`
Displays the makefile and the default makefile, the state file, and hidden dependency reports for the current `make` run.

Several `-f` options indicate the concatenation of the named makefiles.

`-f` *makefile*
`make` uses the named *makefile* (instead of `makefile` or `Makefile`).

`-K` *makestatefile*
If *makestatefile* is a directory, make will write the KEEP_STATE information into a `.make.state` file in that directory. If *makestatefile* is a file, make will write the KEEP_STATE information into the *makestatefile*.

`-p`
: Displays the complete set of macro definitions and target entries.

`-P`
: Displays the complete dependency tree for the default target or the specified target.

Due to its potentially troublesome side effects, it is recommended that you not use the -t (touch) option for `make`.

An option that can be used to shortcut `make` processing is the `-t` option. When run with `-t`, `make` does not perform the rule for building a target. Instead it uses `touch` to alter the modification time for each target that it encounters in the dependency scan. It also updates the state file to reflect what it built. This often creates more problems than it supposedly solves, and it is recommended that you exercise extreme caution if you do use it. Note that if there is no file corresponding to a target entry, `touch` creates it.

clean is the conventional name for a target that removes derived files. It is useful when you want to start a build from scratch.

The following is one example of how *not* to use `make -t`. Suppose you have a target named `clean` that performed housekeeping in the directory by removing target files produced by `make`:

```
clean:
    rm functions main.o data.o
```

If you give the nonsensical command:

```
$ make -t clean
touch clean
$ make clean
'clean' is up to date.
```

you then have to remove the file `clean` before your housekeeping target can work once again.

-q
: Invokes the question mode, and returns a zero or non-zero status code, depending on whether or not the target file is up-to-date.

-r
: Suppresses reading in of the default `makefile` /usr/share/lib/make/make.rules.

-S
: Undoes the effect of the -K option by stopping processing when a non-zero exit status is returned by a command.

Using make to Compile Programs

In previous examples you have seen how to compile a simple C program from a single source file, using both explicit target entries and implicit rules. Most C programs, however, are compiled from several source files. Many include library routines, either from one of the standard system libraries or from a user-supplied library.

Although it may be easier to recompile and link a single-source program using a single `cc` command, it is usually more convenient to compile programs with multiple sources in stages—first, by compiling each source file into a separate object (`.o`) file, and then by linking the object files to form an executable (`a.out`) file. This method requires more disk space, but subsequent (repetitive) recompilations need be performed only on those object files for which the sources have changed, which saves time.

A Simple Makefile

The following makefile is not all that elegant, but it does the job.

```
# Simple makefile for compiling a program from
# two C source files.

.KEEP_STATE:

functions: main.o data.o
    cc -O -o functions main.o data.o
main.o: main.c
    cc -O -c main.c
data.o: data.c
    cc -O -c data.c
clean:
    rm functions main.o data.o
```

Figure 4-3 Simple Makefile for Compiling C Sources: Everything Explicit

In this example, `make` produces the object files `main.o` and `data.o`, and the executable file `functions`:

```
$ make
cc -o functions main.o data.o
cc -O -c main.c
cc -O -c data.c
```

Using `make`'s Predefined Macros

The next example performs exactly the same function, but demonstrates the use of `make`'s predefined macros for the indicated compilation commands. Using predefined macros eliminates the need to edit makefiles when the underlying compilation environment changes. Macros also provide access to the `CFLAGS` macro (and other `FLAGS` macros) for supplying compiler options from the command line. Predefined macros are also used extensively within `make`'s implicit rules. The predefined macros in the following makefile are listed below.[1] They are generally useful for compiling C programs.

Macro names that end in the string FLAGS pass options to a related compiler-command macro. It is good practice to use these macros for consistency and portability. It is also good practice to note the desired default values for them in the makefile.

`COMPILE.c`
The `cc` command line; composed of the values of `CC`, `CFLAGS`, and `CPPFLAGS`, as follows, along with the `–c` option.

```
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -c
```

The root of the macro name, `COMPILE`, is a convention used to indicate that the macro stands for a compilation command line (to generate an object, or `.o` file). The `.c` suffix is a mnemonic device to indicate that the command line applies to `.c` (C source) files.

The complete list of all predefined macros is shown in Table 4-3 on page 178.

`LINK.c`
The basic `cc` command line to link object files, such as `COMPILE.c`, but without the `-c` option and with a reference to the `LDFLAGS` macro:

```
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
```

`CC`
The value `cc`. (You can redefine the value to be the path name of an alternate C compiler.)

`CFLAGS`
Options for the `cc` command; none by default.

`CPPFLAGS`
Options for `cpp`; none by default.

1. Predefined macros are used more extensively than in earlier versions of `make`. Not all of the predefined macros shown here are available with earlier versions.

`LDFLAGS`
Options for the link editor, `ld`; none by default.

```
# Makefile for compiling two C sources
CFLAGS= -O
.KEEP_STATE:

functions: main.o data.o
	$(LINK.c) -o functions main.o data.o
main.o: main.c
	$(COMPILE.c) main.c
data.o: data.c
	$(COMPILE.c) data.c
clean:
	rm functions main.o data.o
```

Figure 4-4 Makefile for Compiling C Sources Using Predefined Macros

Using Implicit Rules to Simplify a Makefile: Suffix Rules

Since the command lines for compiling `main.o` and `data.o` from their `.c` files are now functionally equivalent to the `.c.o` suffix rule, their target entries are redundant; `make` performs the same compilation whether they appear in the makefile or not. This next version of the makefile eliminates them, relying on the `.c.o` rule to compile the individual object files.

```
# Makefile for a program from two C sources
# using suffix rules.
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
	$(LINK.c) -o functions main.o data.o
clean:
	rm functions main.o data.o
```

Figure 4-5 Makefile for Compiling C Sources Using Suffix Rules

A complete list of suffix rules appears in Table 4-2 on page 174.

As `make` processes the dependencies `main.o` and `data.o`, it finds no target entries for them. It checks for an appropriate implicit rule to apply. In this case, `make` selects the `.c.o` rule for building a `.o` file from a dependency file that has the same base name and a `.c` suffix.

make uses the order of appearance in the suffixes list to determine which dependency file and suffix rule to use. For instance, if there were both main.c and main.s files in the directory, make would use the .c.o rule, since .c is ahead of .s in the list.

First, make scans its suffixes list to see if the suffix for the target file appears. In the case of main.o, .o appears in the list. Next, make checks for a suffix rule to build it with, and a dependency file to build it from. The dependency file has the same base name as the target, but a different suffix. In this case, while checking the .c.o rule, make finds a dependency file named main.c, so it uses that rule.

The suffixes list is a special-function target named .SUFFIXES. The various suffixes are included in the definition for the SUFFIXES macro; the dependency list for .SUFFIXES is given as a reference to this macro:

```
SUFFIXES= .o .c .c~ .cc .cc~ .C .C~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln \
    .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ \
    .cps .cps~ .Y .Y~ .L .L~
.SUFFIXES:  $(SUFFIXES)
```

Figure 4-6 The Standard Suffixes List

The following example shows a makefile for compiling a whole set of executable programs, each having just one source file. Each executable is to be built from a source file that has the same basename, and the .c suffix appended. For instance demo_1 is built from demo_1.c.

Like clean, all is a target name used by convention. It builds "all" the targets in its dependency list. Normally, all is the first target; make and make all are usually equivalent.

```
# Makefile for a set of C programs, one source
# per program.  The source file names have ".c"
# appended.
CFLAGS= -O
.KEEP_STATE:

all: demo_1 demo_2 demo_3 demo_4 demo_5
```

In this case, make does not find a suffix match for any of the targets (through demo_5). So, it treats each as if it had a null suffix. It then searches for a suffix rule and dependency file with a valid suffix. In the case of demo_2, it would find a file named demo_2.c. Since there is a target entry for a .c rule, along with a corresponding .c file, make uses that rule to build demo_2 from demo_2.c.

To prevent ambiguity when a target with a null suffix has an explicit dependency, make does not build it using a suffix rule. This makefile

```
program: zap
zap:
```

produces no output:

```
$ make program
$
```

When to Use Explicit Target Entries vs. Implicit Rules

Whenever you build a target from multiple dependency files, you must provide make with an explicit target entry that contains a rule for doing so. When building a target from a single dependency file, it is often convenient to use an implicit rule.

As the previous examples show, make readily compiles a single source file into a corresponding object file or executable. However, it has no built-in knowledge about how to link a list of object files into an executable program. Also, make only compiles those object files that it encounters in its dependency scan. It needs a starting point—a target for which each object file in the list (and ultimately, each source file) is a dependency.

So, for a target built from multiple dependency files, make needs an explicit rule that provides a collating order, along with a dependency list that accounts for its dependency files.

If each of those dependency files is built from just one source, you can rely on implicit rules for them.

Implicit Rules and Dynamic Macros

make maintains a set of macros dynamically, on a target-by-target basis. These macros are used quite extensively, especially in the definitions of implicit rules. It is important to understand what they mean.

Because they aren’t explicitly defined in a makefile, the convention is to document dynamic macros with the $-sign prefix attached (in other words, by showing the macro reference).

They are:

`$@`
: The name of the current target.

`$?`
: The list of dependencies newer than the target.

`$<`
The name of the dependency file, as if selected by `make` for use with an implicit rule.

`$*`
The base name of the current target (the target name stripped of its suffix).

`$%`
For libraries, the name of the member being processed. See “Building Object Libraries” on page 180 for more information.

Implicit rules make use of these dynamic macros in order to supply the name of a target or dependency file to a command line within the rule itself. For instance, in the `.c.o` rule, shown in the next example.

```
.c.o:
    $(COMPILE.c) $< $(OUTPUT_OPTION)
```

`$<` is replaced by the name of the dependency file (in this case the `.c` file) for the current target.

The macro OUTPUT_OPTION has an empty value by default. While similar to CFLAGS in function, it is provided as a separate macro intended for passing an argument to the -o compiler option to force compiler output to a given file name.

In the `.c` rule:

```
.c:
    $(LINK.c) $< -o $@
```

`$@` is replaced with the name of the current target.

Because values for both the `$<` and `$*` macros depend upon the order of suffixes in the suffixes list, you may get surprising results when you use them in an explicit target entry. See “Suffix Replacement in Macro References” on page 183 for a strictly deterministic method for deriving a file name from a related file name.

Dynamic Macro Modifiers

Dynamic macros can be modified by including `F` and `D` in the reference. If the target being processed is in the form of a pathname, `$(@F)` indicates the file name part, while `$(@D)` indicates the directory part. If there are no `/` characters in the target name, then `$(@D)` is assigned the dot character (`.`) as its value. For example, with the target named `/tmp/test`, `$(@D)` has the value `/tmp`; `$(@F)` has the value `test`.

Dynamic Macros and the Dependency List: Delayed Macro References

Dynamic macros are assigned while processing any and all targets. They can be used within the target rule as is, or in the dependency list by prepending an additional `$` character to the reference. A reference beginning with `$$` is called a *delayed* reference to a macro. For instance, the entry:

```
x.o y.o z.o: $$@.BAK
    cp $@.BAK $@
```

could be used to derive `x.o` from `x.o.BAK`, and so forth for `y.o` and `z.o`.

Dependency List Read Twice

This technique works because `make` reads the dependency list twice, once as part of its initial reading of the entire makefile, and again as it processes target dependencies. In each pass through the list, it performs macro expansion. Since the dynamic macros aren't defined in the initial reading, unless references to them are delayed until the second pass, they are expanded to null strings.

The string `$$` is a reference to the predefined macro '`$`'. This macro, conveniently enough, has the value '`$`'; when `make` resolves it in the initial reading, the string `$$@` is resolved to `$@`. In dependency scan, when the resulting `$@` macro reference has a value dynamically assigned to it, `make` resolves the reference to that value.

Note that `make` only evaluates the target-name portion of a target entry in the first pass. A delayed macro reference as a target name will produce incorrect results. The makefile:

```
NONE= none
all: $(NONE)

$$(NONE):
    @: this target's name isn't 'none'
```

produces the following results.

```
$ make
make: Fatal error: Don't know how to make target 'none'
```

Rules Evaluated Once

make evaluates the rule portion of a target entry only once per application of that command, at the time that the rule is executed. Here again, a delayed reference to a make macro will produce incorrect results.

No Transitive Closure for Suffix Rules

There is no transitive closure for suffix rules. If you had a suffix rule for building, say, a .Y file from a .X file, and another for building a .Z file from a .Y file, make would not combine their rules to build a .Z file from a .X file. You must specify the intermediate steps as targets, although their entries may have null rules:

```
trans.Z:
trans.Y:
```

In this example trans.Z will be built from trans.Y if it exists. Without the appearance of trans.Y as a target entry, make might fail with a “don’t know how to build” error, since there would be no dependency file to use. The target entry for trans.Y guarantees that make will attempt to build it when it is out of date or missing. Since no rule is supplied in the makefile, make will use the appropriate implicit rule, which in this case would be the .X.Y rule. If trans.X exists (or can be retrieved from SCCS), make rebuilds both trans.Y and trans.Z as needed.

Adding Suffix Rules

Pattern-matching rules, which are described in “Pattern-Matching Rules:An Alternative to Suffix Rules” on page 173, are often easier to use than suffix rules. The procedure for adding implicit rules is given here for compatibility with previous versions of make.

Although make supplies you with a number of useful suffix rules, you can also add new ones of your own. However, pattern-matching rules, which are described in the next section, are to be preferred when adding new implicit rules. Unless you need to write implicit rules that are compatible with earlier versions of make, you can safely skip the remainder of this section, which describes the traditional method of adding implicit rules to makefiles.

Adding a suffix rule is a two-step process. First, you must add the suffixes of both target and dependency file to the suffixes list by providing them as dependencies to the .SUFFIXES special target. Because dependency lists accumulate, you can add suffixes to the list simply by adding another entry for this target, for example:

```
.SUFFIXES:  .ms .tr
```

Second, you must add a target entry for the suffix rule:

```
.ms.tr:
    troff -t -ms $< > $@
```

A makefile with these entries can be used to format document source files containing ms macros (.ms files) into troff output files (.tr files):

```
$ make doc.tr
troff -t -ms doc.ms > doc.tr
```

Entries in the suffixes list are contained in the SUFFIXES macro. To insert suffixes at the head of the list, first clear its value by supplying an entry for the .SUFFIXES target that has no dependencies. This is an exception to the rule that dependency lists accumulate. You can clear a previous definition for this target by supplying a target entry with no dependencies and no rule like this:

```
.SUFFIXES:
```

You can then add another entry containing the new suffixes, followed by a reference to the SUFFIXES macro, as shown below.

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Pattern-Matching Rules:An Alternative to Suffix Rules

A *pattern-matching rule* is similar to an implicit rule in function. Pattern-matching rules are easier to write, and more powerful, because you can specify a relationship between a target and a dependency based on prefixes (including path names) and suffixes, or both. A pattern-matching rule is a target entry of the form:

```
tp%ts:  dp%ds
            rule
```

where *tp* and *ts* are the optional prefix and suffix in the target name, *dp* and *ds* are the (optional) prefix and suffix in the dependency name, and `%` is a wild card that stands for a base name common to both.

`make` checks for pattern-matching rules ahead of suffix rules. While this allows you to override the standard implicit rules, it is not recommended.

If there is no rule for building a target, `make` searches for a pattern-matching rule, *before* checking for a suffix rule. If `make` can use a pattern-matching rule, it does so.

If the target entry for a pattern-matching rule contains no rule, `make` processes the target file as if it had an explicit target entry with no rule; `make` therefore searches for a suffix rule, attempts to retrieve a version of the target file from SCCS, and finally, treats the target as having a null rule (flagging that target as updated in the current run).

A pattern-matching rule for formatting a `troff` source file into a `troff` output file looks like:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

`make`'s *Default Suffix Rules and Predefined Macros*

The following tables show the standard set of suffix rules and predefined macros supplied to `make` in the default makefile, `/usr/share/lib/make/make.rules`.

Table 4-2 Standard Suffix Rules

Use	Suffix Rule Name	Command Line(s)
Assembly Files	`.s.o`	`$(COMPILE.s) –o $@ $<`
	`.s`	`$(COMPILE.s) –o $@ $<`
	`.s.a`	`$(COMPILE.s) –o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.S.o`	`$(COMPILE.S) –o $@ $<`
	`.S.a`	`$(COMPILE.S) –o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
C Files (.c Rules)	`.c`	`$(LINK.c) –o $@ $< $(LDLIBS)`
	`.c.ln`	`$(LINT.c) $(OUTPUT_OPTION) –i $<`
	`.c.o`	`$(COMPILE.c) $(OUTPUT_OPTION) $<`
	`.c.a`	`$(COMPILE.c) –o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
C++ Files	`.cc`	`$(LINK.cc) -o $@ $< $(LDLIBS)`
	`.cc.o`	`$(COMPILE.cc) $(OUTPUT_OPTION) $<`
	`.cc.a`	`$(COMPILE.cc) -o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`

Table 4-2 Standard Suffix Rules *(Continued)*

Use	Suffix Rule Name	Command Line(s)
C++ Files (SVr4 style)	`.C`	`$(LINK.C) -o $@ $< $(LDFLAGS) $*.c`
	`.C.o`	`$(COMPILE.C) $<`
	`.C.a`	`$(COMPILE.C) $<`
		`$(AR)  $(ARFLAGS) $@ $*.o`
		`$(RM) -f $*.o`
FORTRAN 77 Files	`.cc.o`	`$(LINK.f) –o $@ $< $(LDLIBS)`
	`.cc.a`	`$(COMPILE.f) $(OUTPUT_OPTION) $<`
		`$(COMPILE.f) –o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.F`	`$(LINK.F) –o $@ $< $(LDLIBS)`
	`.F.o`	`$(COMPILE.F) $(OUTPUT_OPTION) $<`
	`.F.a`	`$(COMPILE.F) –o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`

Table 4-2 Standard Suffix Rules *(Continued)*

Use	Suffix Rule Name	Command Line(s)
lex Files	.l	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(LINK.c) -o $@ $*.c $(LDLIBS)
		$(RM) $*.c
	.l.c	$(RM) $@
		$(LEX.l) $< > $@
	.l.ln	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(LINT.c) -o $@ -i $*.c
		$(RM) $*.c
	.l.o	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(COMPILE.c) -o $@ $*.c
		$(RM) $*.c
	.L.C	$(LEX) $(LFLAGS) $<
	.L.o	$(LEX)(LFLAGS) $<
		$(COMPILE.C) lex.yy.c
	.L.o	rm -f lex.yy.c
		mv lex.yy.o $@
Modula 2 Files	.mod	$(COMPILE.mod) -o $@ -e $@ $<
	.mod.o	$(COMPILE.mod) -o $@ $<
	.def.sym	$(COMPILE.def) -o $@ $<
NeWS	.cps.h	$(CPS) $(CPSFLAGS) $*.cps
Pascal Files	.p	$(LINK.p) -o $@ $< $(LDLIBS)
	.p.o	$(COMPILE.p) $(OUTPUT_OPTION) $<

Table 4-2 Standard Suffix Rules *(Continued)*

Use	Suffix Rule Name	Command Line(s)
Ratfor Files	`.r`	`$(LINK.r) –o $@ $< $(LDLIBS)`
	`.r.o`	`$(COMPILE.r) $(OUTPUT_OPTION) $<`
	`.r.a`	`$(COMPILE.r) –o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
Shell Scripts	`.sh`	`$(RM) $@`
		`cat $< >$@`
		`chmod +x $@`
yacc Files (.y.c Rules)	`.y`	`$(YACC.y) $<`
		`$(LINK.c) –o $@ y.tab.c $(LDLIBS)`
		`$(RM) y.tab.c`
	`.y.c`	`$(YACC.y) $<`
		`mv y.tab.c $@`
	`.y.ln`	`$(YACC.y) $<`
		`$(LINT.c) –o $@ –i y.tab.c`
		`$(RM) y.tab.c`
	`.y.o`	`$(YACC.y) $<`
		`$(COMPILE.c) –o $@ y.tab.c`
		`$(RM) y.tab.c`
yacc Files (SVr4)	`.Y.C`	`$(YACC) $(YFLAGS) $<`
		`mv y.tab.c $@`
	`.Y.o`	`$(YACC) $(YFLAGS) $<`
		`$(COMPILE.c) y.tab.c`
		`rm -f y.tab.c`
		`mv y.tab.o $@`

Table 4-3 Predefined and Dynamic Macros

Use	Macro	Default Value
Library Archives	`AR`	`ar`
	`ARFLAGS`	`rv`
Assembler Commands	`AS`	`as`
	`ASFLAGS`	
	`COMPILE.s`	`$(AS) $(ASFLAGS)`
	`COMPILE.S`	`$(CC) $(ASFLAGS) $(CPPFLAGS) -target -c`
C Compiler Commands	`CC`	`cc`
	`CFLAGS`	
	`CPPFLAGS`	
	`COMPILE.c`	`$(CC) $(CFLAGS) $(CPPFLAGS) –c`
	`LINK.c`	`$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)`
C++ Compiler Commands[1]	`CCC`	`CC`
	`CCFLAGS`	
	`COMPILE.cc`	`$(CCC) $(CCFLAGS) $(CPPFLAGS) –c`
	`LINK.cc`	`$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)`
C++ SVr4 Compiler Commands	`(C++C)`	`CC`
	`(C++FLAGS)`	`-O`
	`COMPILE.C`	`$(C++C) $(C++FLAGS) $(CPPFLAGS) –c`
	`LINK.C`	`$(C++C) $(C++FLAGS) $(CPPFLAGS) $(LDFLAGS) -target`
FORTRAN 77 Compiler Commands	`FC in SVr4`	`f77`
	`FFLAGS`	
	`COMPILE.f`	`$(FC) $(FFLAGS) –c`
	`LINK.f`	`$(FC) $(FFLAGS) $(LDFLAGS)`
	`COMPILE.F`	`$(FC) $(FFLAGS) $(CPPFLAGS) –c`
	`LINK.F`	`$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS)`

Table 4-3 Predefined and Dynamic Macros *(Continued)*

Use	Macro	Default Value
Link Editor Command	LD	ld
	LDFLAGS	
lex Command	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint Command	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS)
Modula 2 Commands	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS)
NeWS	CPS	cps
	CPSFLAGS	
Pascal Compiler Commands	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS)
Ratfor Compilation Commands	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(LDFLAGS)
rm Command	RM	rm -f

Table 4-3 Predefined and Dynamic Macros *(Continued)*

Use	Macro	Default Value
`yacc` Command	`YACC`	`yacc`
	`YFLAGS`	
	`YACC.y`	`$(YACC) $(YFLAGS)`
Suffixes List	`SUFFIXES`	`.o .c .c~ .cc .cc~ .C .C~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .cps .cps~ .Y .Y~ .L .L~`
`SCCS get` Command	`.SCCS_GET`	`sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@ -G$@`
	`SCCSGETFLAGS`	`-s`

1. For backward compatibility, the C++ macros have alternate forms. For `C++C`, you can instead use `CCC`; instead of `C++FLAGS`, you can use `CCFLAGS`; for `COMPILE.C`, you can use `COMPILE.cc`; and `LINK.cc` can be substituted for `LINK.C`. Note that these alternate forms will disappear for future releases.

Building Object Libraries

Libraries, Members, and Symbols

An object library is a set of object files contained in an `ar` library archive.[1] Various languages make use of object libraries to store compiled functions of general utility, such as those in the C library.

`ar` reads in a set of one or more files to create a library. Each member contains the text of one file, preceded by a header. The member header contains information from the file directory entry, including the modification time. This allows `make` to treat the library member as a separate entity for dependency checking.

When you compile a program that uses functions from an object library (specifying the proper library either by filename, or with the `-l` option to `cc`), the link editor selects and links with the library member that contains a needed symbol.

1. See *ar(1)* and *lorder(1)* in the *SunOS Reference Manual* for details about library archive files.

You can use `ar` to generate a symbol table for a library of object files. `ld` requires this table in order to provide random access to symbols within the library—to locate and link object files in which functions are defined. You can also use `lorder` and `tsort` ahead of time to put members in calling order within the library. (See `ar(1) and lorder`(1) for details.) For very large libraries, it is a good idea to do both.

Library Members and Dependency Checking

`make` recognizes a target or dependency of the form:

lib.a(*member* . . .)

as a reference to a library member, or a space-separated list of members.[1] In this version of `make`, all members in a parenthesized list are processed. For example, the following target entry indicates that the library named `librpn.a` is built from members named `stacks.o` and `fifos.o`. The pattern-matching rule indicates that each member depends on a corresponding object file, and that object file is built from its corresponding source file using an implicit rule.

```
librpn.a:   librpn.a (stacks.o fifos.o)
       ar rv $@ $?

 $@
librpn.a (%.o): %.o
       @true
```

When used with library-member notation, the dynamic macro `$?` contains the list of files that are newer than their corresponding members:

```
$ make
cc -c stacks.c
cc -c fifos.c
ar rv librpn.a stacks.o fifos.o
a - stacks.o
a - fifos.o
```

1. Earlier versions of `make` recognize this notation. However, only the first item in a parenthesized list of members was processed.

Libraries and the `$%` *Dynamic Macro*

The `$%` dynamic macro is provided specifically for use with libraries. When a library member is the target, the member name is assigned to the `$%` macro. For instance, given the target `libx.a(demo.o)` the value of `$%` would be `demo.o`.

`.PRECIOUS`*: Preserving Libraries Against Removal due to Interrupts*

Normally, if you interrupt `make` in the middle of a target, the target file is removed. For individual files this is a good thing, otherwise incomplete files with brand new modification times might be left in the directory. For libraries that consist of several members, the story is different. It is often better to leave the library intact, even if one of the members is still out-of-date. This is especially true for large libraries, especially since a subsequent `make` run will pick up where the previous one left off—by processing the object file or member whose processing was interrupted.

`.PRECIOUS` is a special target that is used to indicate which files should be preserved against removal on interrupts; `make` does not remove targets that are listed as its dependencies. If you add the line:

```
.PRECIOUS:  librpn.a
```

to the makefile shown above, run `make`, and interrupt the processing of `librpn.a`, the library is preserved.

Using `make` *to Maintain Libraries and Programs*

In previous sections you learned how `make` can help compile simple programs and build simple libraries. This section describes some of `make`’s more advanced features for maintaining complex programs and libraries.

More about Macros

Macro definitions can appear on any line in a makefile; they can be used to abbreviate long target lists or expressions, or as shorthand to replace long strings that would otherwise have to be repeated.

You can even use macros to derive lists of object files from a list of source files. Macro names are allocated as the makefile is read in; the value a particular macro reference takes depends upon the most recent value assigned.[1] With the exception of conditional and dynamic macros, `make` assigns values in the order the definitions appear.

Embedded Macro References

Macro references can be embedded within other references.[2]

```
$(CPPFLAGS$(TARGET_ARCH))
```

The += assignment appends the indicated string to any previous value for the macro.

In which case they are expanded from innermost to outermost. With the following definitions, `make` will supply the correct symbol definition for (for example) a Sun-4 system.

```
CPPFLAGS-sun4 = -DSUN4
CPPFLAGS += $(CPPFLAGS-$(TARGET_ARCH))
```

Suffix Replacement in Macro References

`make` provides a mechanism for replacing suffixes of words that occur in the value of the referred-to macro.[3] A reference of the form:

`$(`*macro*:*old-suffix*=*new-suffix*`)`

is a *suffix replacement* macro reference. You can use a such a reference to express the list of object files in terms of the list of sources:

```
OBJECTS= $(SOURCES:.c=.o)
```

1. Actually, macro evaluation is a bit more complicated than this. Refer to “Passing Parameters to Nested make Commands” on page 197 for more information.
2. Not supported in previous versions of `make`.
3. Although conventional suffixes start with dots, a suffix may consist of any string of characters.

In this case, make replaces all occurrences of the .c suffix in words within the value with the .o suffix. The substitution is not applied to words that do not end in the suffix given. The following makefile:

```
SOURCES= main.c data.c moon
OBJECTS= $(SOURCES:.c=.o)

all:
    @echo $(OBJECTS)
```

illustrates this very simply:

```
$ make
main.o data.o moon
```

Using lint *with* make

For easier debugging and maintenance of your C programs use the lint tool. lint also checks for C constructs that are not considered portable across machine architectures. It can be a real help in writing portable C programs.

lint, the C program verifier, is an important tool for forestalling the kinds of bugs that are most difficult and tedious to track down. These include uninitialized pointers, parameter-count mismatches in function calls, and non-portable uses of C constructs. As with the clean target, lint is a target name used by convention; it is usually a good practice to include it in makefiles that build C programs. lint produces output files that have been preprocessed through cpp and its own first (parsing) pass. These files characteristically end in the .ln suffix[1] and can also be derived from the list of sources through suffix replacement:

```
LINTFILES= $(SOURCES:.c=.ln)
```

A target entry for the lint target might appear as:

```
lint: $(LINTFILES)
    $(LINT.c) $(LINTFILES)
$(LINTFILES):
    $(LINT.c) $@ -i
```

1. This may not be true for some versions of lint.

There is an implicit rule for building each `.ln` file from its corresponding `.c` file, so there is no need for target entries for the `.ln` files. As sources change, the `.ln` files are updated whenever you run

```
make lint
```

Since the `LINT.c` predefined macro includes a reference to the `LINTFLAGS` macro, it is a good idea to specify the `lint` options to use by default (none in this case). Since `lint` entails the use of `cpp`, it is a good idea to use `CPPFLAGS`, rather than `CFLAGS` for compilation preprocessing options (such as `-I`). The `LINT.c` macro does not include a reference to `CFLAGS`.

Also, when you run `make clean`, you will want to get rid of any `.ln` files produced by this target. It is a simple enough matter to add another such macro reference to a `clean` target.

Linking with System-Supplied Libraries

The next example shows a makefile that compiles a program that uses the `curses` and `termlib` library packages for screen-oriented cursor motion.

```
# Makefile for a C program with curses and termlib.

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
    $(LINK.c) -o $@ main.o data.o -lcurses -ltermlib
lint: main.ln data.ln
    $(LINT.c) main.ln data.ln
main.ln data.ln:
    $(LINT.c) $@ -i
clean:
    rm -f functions main.o data.o main.ln data.ln
```

Figure 4-7 Makefile for a C Program with System-Supplied Libraries

Since the link editor resolves undefined symbols as they are encountered, it is normally a good idea to place library references at the end of the list of files to link.

This makefile produces:

```
$ make
cc -O -c main.c
cc -O -c data.c
cc -O -o functions main.o data.o -lcurses -ltermlib
```

Compiling Programs for Debugging and Profiling

Compiling programs for debugging or profiling introduces a new twist to the procedure and to the makefile. These variants are produced from the same source code, but are built with different options to the C compiler. The `cc` option to produce object code that is suitable for debugging is `-g`. The `cc` options that produce code for profiling are `-O` and `-pg`.

Since the compilation procedure is the same otherwise, you *could* give `make` a definition for `CFLAGS` on the command line. Since this definition overrides the definition in the makefile, and `.KEEP_STATE` assures any command lines affected by the change are performed, the following `make` command produces the results as presented in this example:

```
$ make "CFLAGS= -O -pg"
cc -O -pg -c main.c
cc -O -pg -c data.c
cc -O -pg -o functions main.o data.o -lcurses -ltermlib
```

Of course, you may not want to memorize these options or type a complicated command like this, especially when you can put this information in the makefile. What is needed is a way to tell `make` how to produce a debugging or profiling variant, and some instructions in the makefile that tell it how. One way to do this might be to add two new target entries, one named `debug`, and the other named `profile`, with the proper compiler options hard-coded into the command line.

A better way would be to add these targets, but rather than hard-coding their rules, include instructions to alter the definition of `CFLAGS` depending upon which target it starts with. Then, by making each one depend on the existing target for `functions`, `make` could simply make use of its rule, along with the specified options.

Instead of saying:

```
    make "CFLAGS= -g"
```

to compile a variant for debugging, you could say:

```
make debug
```

The question is, how do you tell `make` that you want a macro defined one way for one target (and its dependencies), and another way for a different target?

Conditional Macro Definitions

A conditional macro definition is a line of the form:

target-list `:=` *macro* `=` *value*

Each word in ***target-list*** may contain one % pattern; `make` must know which targets the definition applies to, so you can't use a conditional macro definition to alter a target name.

which assigns the given *value* to the indicated *macro* while `make` is processing the target named *target-name* and its dependencies. The following lines give `CFLAGS` an appropriate value for processing each program variant.

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

Note that when you use a reference to a conditional macro in the dependency list, that reference must be delayed (by prepending a second `$`). Otherwise, `make` will expand the reference before the correct value has been assigned. When it encounters a (possibly) incorrect reference of this sort, `make` issues a warning.

Compiling Debugging and Profiling Variants

The following makefile produces optimized, debugging, or profiling variants of a C program, depending on which target you specify (the default is the optimized variant). Command dependency checking guarantees that the program and its object files will be recompiled whenever you switch between variants.

```
# Makefile for a C program with alternate
# debugging and profiling variants.

CFLAGS= -O

.KEEP_STATE:

all debug profile: functions

debug := CFLAGS = -g
profile := CFLAGS = -pg -O

functions: main.o data.o
    $(LINK.c) -o $@ main.o data.o -lcurses -ltermlib
lint: main.ln data.ln
    $(LINT.c) main.ln data.ln
clean:
    rm -f functions main.o data.o main.ln data.ln
```

Figure 4-8 Makefile for a C Program with Alternate Debugging and Profiling Variants

The first target entry specifies three targets, starting with `all`.

Debugging and profiling variants aren't normally considered part of a finished program.

`all` traditionally appears as the first target in makefiles with alternate starting targets (or those that process a list of targets). Its dependencies are "all" targets that go into the final build, whatever that may be. In this case, the final variant is optimized. The target entry also indicates that `debug` and `profile` depend on `functions` (the value of `$(PROGRAM)`).

The next two lines contain conditional macro definitions for `CFLAGS`.

Next comes the target entry for `functions`. When `functions` is a dependency for `debug`, it is compiled with the `-g` option.

The next example applies a similar technique to maintaining a C object library.

```
# Makefile for a C library with alternate variants.

CFLAGS= -O

.KEEP_STATE
.PRECIOUS:  libpkg.a

all debug profile: libpkg.a
debug := CFLAGS= -g
profile := CFLAGS= -pg -O

libpkg.a: libpkg.a(calc.o map.o draw.o)
    ar rv $@ $?
    libpkg.a(%.o): %.o
    @true
lint: calc.ln map.ln draw.ln
    $(LINT.c) calc.ln map.ln draw.ln
clean:
    rm -f libpkg.a calc.o map.o draw.o calc.ln \
    map.ln draw.ln
```

Figure 4-9 Makefile for a C Library with Alternate Variants

Maintaining Separate Program and Library Variants

The previous two examples are adequate when development, debugging, and profiling are done in distinct phases. However, they suffer from the drawback that all object files are recompiled whenever you switch between variants, which can result in unnecessary delays. The next two examples illustrate how all three variants can be maintained as separate entities.

To avoid the confusion that might result from having three variants of each object file in the same directory, you can place the debugging and profiling object files and executables in subdirectories. However, this requires a technique for adding the name of the subdirectory as a prefix to each entry in the list of object files.

Pattern-Replacement Macro References

A pattern-replacement macro reference is similar in form and function to a suffix replacement reference.[1] You can use a pattern-replacement reference to add or alter a prefix, suffix, or both, to matching words in the value of a macro. A pattern-replacement reference takes the form:

`$(`*macro*`:`*p*%*s* =*np*%*ns*`)`

where *p* is the existing prefix to replace (if any), *s* is the existing suffix to replace (if any), *np* and *ns* are the new prefix and new suffix, and `%` is a wild card. The pattern replacement is applied to all words in the value that match '*p*`%`*s*'. For instance:

```
SOURCES= old_main.c old_data.c moon
OBJECTS= $(SOURCES:old_%.c=new_%.o)
all:
    @echo $(OBJECTS)
```

produces:

```
$ make
new_main.o new_data.o moon
```

You may use any number of `%` wild cards in the right-hand (replacement) side of the = sign, as needed. The following replacement:

```
...
OBJECTS= $(SOURCES:old_%.c=%/%.o)
```

would produce:

```
main/main.o data/data.o moon
```

1. As with pattern-matching rules, pattern-replacement macro references aren't available in earlier versions of `make`.

Note, however, that pattern-replacement macro references should not appear in the dependency line of the target entry for a pattern-matching rule. This produces a conflict, since `make` cannot tell whether the wild card applies to the macro, or to the target (or dependency) itself. With the makefile:

```
OBJECT= .o

x:
x.Z:
    @echo correct
%: %$(OBJECT:%o=%Z)
```

it looks as if `make` should attempt to build `x` from `x.Z`. However, the pattern-matching rule is not recognized; `make` cannot determine which of the `%` characters in the dependency line to use in the pattern-matching rule.

Makefile for a Program with Separate Variants

`make` performs the rule in the .INIT target just after the makefile is read.

The following example shows a makefile for a C program with separately maintained variants. First, the `.INIT` special target creates the `debug_dir` and `profile_dir` subdirectories (if they don't already exist), which will contain the debugging and profiling object files and executables.

The variant executables are made to depend on the object files listed in the `VARIANTS.o` macro. This macro is given the value of `OBJECTS` by default; later on it is reassigned using a conditional macro definition, at which time either the `debug_dir/` or `profile_dir/` prefix is added. Executables in the subdirectories depend on the object files that are built in those same subdirectories.

Next, pattern-matching rules are added to indicate that the object files in both subdirectories depend upon source (`.c`) files in the working directory. This is the key step needed to allow all three variants to be built and maintained from a single set of source files.

Finally, the `clean` target has been updated to recursively remove the `debug_dir` and `profile_dir` subdirectories and their contents, which should be regarded as temporary. This is in keeping with the custom that derived files are to be built in the same directory as their sources, since the subdirectories for the variants are considered temporary.

```
# Simple makefile for maintaining separate debugging and
# profiling program variants.

CFLAGS= -O

SOURCES= main.c rest.c
OBJECTS= $(SOURCES:%.c=$(VARIANT)/%.o)
VARIANT= .

functions profile debug: $$(OBJECTS)
    $(LINK.c) -o $(VARIANT)/$@ $(OBJECTS)

debug := VARIANT = debug_dir
debug := CFLAGS = -g
profile := VARIANT = profile_dir
profile := CFLAGS = -O -pg

.KEEP_STATE:
.INIT:  profile_dir debug_dir
profile_dir debug_dir:
    test -d $@ || mkdir $@
$$(VARIANT)/%.o: %.c
    $(COMPILE.c) $< -o $@
clean:
    rm -r profile_dir debug_dir $(OBJECTS) functions
```

Figure 4-10 Sample Makefile for Separate Debugging and Profiling Program Variants

Makefile for a Library with Separate Variants

The modifications for separate library variants are quite similar:

```
# Makefile for maintaining separate library variants.

CFLAGS= -O

SOURCES= main.c rest.c
LIBRARY= lib.a
LSOURCES= fnc.c

OBJECTS= $(SOURCES:%.c=$(VARIANT)/%.o)
VLIBRARY= $(LIBRARY:%.a=$(VARIANT)/%.a)
LOBJECTS= $(LSOURCES:%.c=$(VARIANT)/%.o)
VARIANT= .

program profile debug: $$(OBJECTS) $$(VLIBRARY)
    $(LINK.c) -o $(VARIANT)/$@ $<

lib.a debug_dir/lib.a profile_dir/lib.a: $$(LOBJECTS)
    ar rv $@ $?

$$(VLIBRARY)($$(VARIANT)%.o): $$(VARIANT)%.o
    @true
profile := VARIANT = profile_dir
profile := CFLAGS = -O -pg

debug := VARIANT = debug_dir
debug := CFLAGS = -g

.KEEP_STATE:
profile_dir debug_dir:
    test -d $@ || mkdir $@
$$(VARIANT)/%.o: %.c
    $(COMPILE.c) $< -o $@
```

Figure 4-11 Sample Makefile for Separate Debugging and Profiling Library Variants

While an interesting and useful compilation technique, this method for maintaining separate variants is a bit complicated. For the sake of clarity, it is omitted from subsequent examples.

Maintaining a Directory of Header Files

The makefile for maintaining an `include` directory of headers is really quite simple. Since headers consist of plain text, all that is needed is a target, `all`, that lists them as dependencies. Automatic SCCS retrieval takes care of the rest. If you use a macro for the list of headers, this same list can be used in other target entries.

```
# Makefile for maintaining an include directory.

FILES.h= calc.h map.h draw.h

all: $(FILES.h)

clean:
    rm -f $(FILES.h)
```

Compiling and Linking with Your Own Libraries

When preparing your own library packages, it makes sense to treat each library as an entity that is separate from its header(s) and the programs that use it. Separating programs, libraries, and headers into distinct directories often makes it easier to prepare makefiles for each type of module. Also, it clarifies the structure of a software project.

It is not a good idea to have things pop up all over the file system as a result of running `make`.

A courteous and necessary convention of makefiles is that they only build files in the working directory, or in temporary subdirectories. Unless you are using `make` specifically to install files into a specific directory on an agreed-upon file system, it is regarded as very poor form for a makefile to produce output in another directory.

Building programs that rely on libraries in other directories adds several new wrinkles to the makefile. Up until now, everything needed has been in the directory, or else in one of the standard directories that are presumed to be stable. This is not true for user-supplied libraries that are part of a project under development.

Since these libraries aren’t built automatically (there is no equivalent to hidden dependency checking for them), you must supply target entries for them. On the one hand, you need to ensure the libraries you link with are up to date.

On the other, you need to observe the convention that a makefile should only maintain files in the local directory. In addition, the makefile should not contain information duplicated in another.

Nested make *Commands*

The solution is to use a nested make command, running in the directory the library resides in, to rebuild it (according to the target entry in the makefile there).

The MAKE macro, which is set to the value "make" by default, overrides the -n option. Any command line in which it is referred to is executed, even though -n may be in effect. Since this macro is used to invoke make, and since the make it invokes inherits -n from the special MAKEFLAGS macro, make can trace a hierarchy of nested make commands with the -n option.

```
# First cut entry for target in another directory.

../lib/libpkg.a:
    cd ../lib ; $(MAKE) libpkg.a
```

The library is specified with a path name relative to the current directory. In general, it is better to use relative path names. If the project is moved to a new root directory or machine, so long as its structure remains the same relative to that new root directory, all the target entries will still point to the proper files.

Within the nested make command line, the dynamic macro modifiers F and D come in handy, as does the MAKE predefined macro. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value.

The target entry can be rewritten as:

```
# Second cut.

../lib/libpkg.a:
    cd $(@D); $(MAKE) $(@F)
```

Forcing A Nested make *Command to Run*

Because it has no dependencies, this target will only run when the file named ../lib/libpkg.a is missing. If the file is a library archive protected by .PRECIOUS, this could be a rare occurrence. The current make invocation neither knows nor cares about what that file depends on, nor should it. It is the nested invocation that decides whether and how to rebuild that file.

After all, just because a file is present in the file system doesn't mean it is up-to-date. This means that you have to force the nested `make` to run, regardless of the presence of the file, by making it depend on another target with a null rule (and no extant file):

```
# Reliable target entry for a nested make command.

../lib/libpkg.a:  FORCE
    cd $(@D); $(MAKE) $(@F)
FORCE:
```

Figure 4-12 Target Entry for a Nested `make` Command

In this way, `make` reliably changes to the correct directory `../lib` and builds `libpkg.a` if necessary, using instructions from the makefile found in that directory.

These lines are produced by the nested `make` run.

```
$ make ../lib/libpkg.a
cd ../lib; make libpkg.a
make libpkg.a
'libpkg.a' is up to date.
```

The following makefile uses a nested `make` command to process local libraries that a program depends on.

```
# Makefile for a C program with user-supplied  libraries and
# nested make commands.

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o ../lib/libpkg.a
    $(LINK.c) -o $@ main.o data.o ../lib/libpkg.a -lcurses -ltermlib
../lib/libpkg.a:  FORCE
    cd $(@D); $(MAKE) $(@F)
FORCE:

lint: main.ln data.ln
    $(LINT.c) main.ln data.ln
clean:
    rm -f functions main.o data.o main.ln data.ln
```

Figure 4-13 Makefile for C Program with User-Supplied Libraries

When ../lib/libpkg.a is up to date, this makefile produces:

```
$ make
cc -O -c main.c
cc -O -c data.c
cd ../lib; make libpkg.a
'libpkg.a' is up to date.
cc -O -o functions main.o data.o ../lib/libpkg.a -lcurses -l   termlib
```

The MAKEFLAGS *Macro*

Do not define MAKEFLAGS in your makefiles.

Like the MAKE macro, MAKEFLAGS is also a special case. It contains flags (that is, single-character options) for the make command. Unlike other FLAGS macros, the MAKEFLAGS value is a concatenation of flags, without a leading '-'. For instance the string eiknp would be a recognized value for MAKEFLAGS, while -f x.mk or macro=value would not.

If the MAKEFLAGS environment variable is set, make runs with the combination of flags given on the command line and contained in that variable.

The value of MAKEFLAGS is always exported, whether set in the environment or not, and the options it contains are passed to any nested make commands (whether invoked by $(MAKE), make, or /usr/bin/make). This insures that nested make commands are always passed the options which the parent make was invoked.

Passing Parameters to Nested make *Commands*

With the exception of MAKEFLAGS,[1] make imports variables from the environment and treats them as if they were defined macros. In turn, make propagates those environment variables and their values to commands it invokes, including nested make commands. Macros can also be defined as command-line arguments, as well as the makefile. This can lead to name-value conflicts when a macro is defined in more than one place, and make has a fairly complicated precedence rule for resolving them.

First, conditional macro definitions always take effect within the targets (and their dependencies) for which they are defined.

1. and SHELL. The SHELL environment variable is neither imported nor exported in this version of make.

If make is invoked with a macro-definition argument, that definition takes precedence over definitions given either within the makefile, or imported from the environment. (This does not necessarily hold true for nested make commands, however.) Otherwise, if you define (or redefine) a macro within the makefile, the most recent definition applies. The latest definition normally overrides the environment. Lastly, if the macro is defined in the default file and nowhere else, that value is used.

With nested make commands, definitions made in the makefile normally override the environment, but only for the makefile in which each definition occurs; the value of the corresponding environment variable is propagated regardless.

Command-line definitions override both environment and makefile definitions, but only in the make run for which they are supplied. Although values from the command line are propagated to nested make commands, they are overridden both by definitions in the nested makefiles, and by environment variables imported by the nested make commands.

The -e option behaves more consistently. The environment overrides macro definitions made in any makefile, and command-line definitions are always used ahead of definitions in the makefile and the environment. One drawback to -e is that it introduces a situation in which information that is *not contained in the makefile* can be critical to the success or failure of a build.

To avoid these complications, when you want to pass a specific value to an entire hierarchy of make commands, run make -e in a subshell with the environment set properly (in the C shell):

```
% (unsetenv MAKEFLAGS LDFLAGS; setenv CFLAGS -g ; make -e )
```

If you want to test out the cases yourself, you can use the following makefiles to illustrate the various cases.

```
# top.mk

MACRO= "Correct but unexpected."

top:
    @echo "------------------------------ top"
    echo $(MACRO)
    @echo "------------------------------"
    $(MAKE) -f nested.mk
    @echo "------------------------------ clean"
clean:
    rm nested
```

```
# nested.mk

MACRO=nested

nested:
    @echo "------------------------------ nested"
    touch nested
    echo $(MACRO)
    $(MAKE) -f top.mk
    $(MAKE) -f top.mk clean
```

The following is a summary of macro assignment orders:

Table 4-4 Summary of Macro Assignment Order

Without `-e`	*With* `-e` *in effect*
top-level `make` *commands*:	
Conditional definitions	Conditional definitions
`Make` command line	`Make` command line
Latest makefile definition	Environment value
Environment value	Latest makefile definition
Predefined value, if any	Predefined value, if any
nested `make` *commands:*	
Conditional definitions	Conditional definitions

Table 4-4 Summary of Macro Assignment Order

`Make` command line	`Make` command line
Latest makefile definition	Parent `make` cmd. line
Environment variable	Environment value
Predefined value, if any	Latest makefile definition
Parent `make` cmd. line	Predefined value, if any

Compiling Other Source Files

Compiling and Linking a C Program with Assembly Language Routines

The makefile in the next example maintains a program with C source files linked with assembly language routines. There are two varieties of assembly source files: those that do not contain `cpp` preprocessor directives, and those that do.

By convention, assembly source files without preprocessor directives have the `.s` suffix. Assembly sources that require preprocessing have the `.S` suffix.

ASFLAGS passes options for as to the `.s.o` and `.S.o` implicit rules.

Assembly sources are assembled to form object files in a fashion similar to that used to compile C sources. The object files can then be linked into a C program. `make` has implicit rules for transforming `.s` and `.S` files into object files, so a target entry for a C program with assembly routines need only specify how to link the object files. You can use the familiar `cc` command to link object files produced by the assembler:

```
CFLAGS= -O
ASFLAGS= -O

.KEEP_STATE:

driver: c_driver.o s_routines.o S_routines.o
    cc -o driver c_driver.o s_routines.o S_routines.o
```

Figure 4-14 Summary of Macro Assignment Order

Note that the `.S` files are processed using the `cc` command, which invokes the C preprocessor `cpp`, and invokes the assembler.

Compiling lex *and* yacc *Sources*

lex and yacc produce C source files as output. Source files for lex end in the suffix .l, while those for yacc end in .y. When used separately, the compilation process for each is similar to that used to produce programs from C sources alone.

There are implicit rules for compiling the lex or yacc sources into .c files; from there, the files are further processed with the implicit rules for compiling object files from C sources. When these source files contain no #include statements, there is no need to keep the .c file, which in this simple case serves as an intermediate file. In this case one could use .l.o rule, or the .y.o rule, to produce the object files, and remove the (derived) .c files.

For example, the makefile:

```
CFLAGS= -O
.KEEP_STATE:

all: scanner parser
scanner: scanner.o
parser: parser.o
```

produces the result shown below.

```
$ make -n
rm -f scanner.c
lex -t scanner.l > scanner.c
cc -O -c -o scanner.o scanner.c
rm -f scanner.c
yacc parser.y
cc -O -c -o parser.o y.tab.c
rm -f y.tab.c
```

Things get to be a bit more complicated when you use lex and yacc in combination. In order for the object files to work together properly, the C code from lex must include a header produced by yacc. It may be necessary to recompile the C source file produced by lex when the yacc source file changes. In this case, it is better to retain the intermediate (.c) files produced by lex, as well as the additional .h file that yacc provides, to avoid running lex whenever the yacc source changes.

The following makefile maintains a program built from a `lex` source, a `yacc` source, and a C source file.

yacc produces output files named y.tab.c and y.tab.h. If you want the output files to have the same basename as the source file, you must rename them.

```
CFLAGS= -O
.KEEP_STATE:

a2z: c_functions.o scanner.o parser.o
    cc -o $@ c_functions.o scanner.o parser.o
scanner.c:

parser.c + parser.h: parser.y
    yacc -d parser.y
    mv y.tab.c parser.c
    mv y.tab.h parser.h
```

Since there is no transitive closure for implicit rules, you must supply a target entry for `scanner.c`. This entry bridges the gap between the `.l.c` implicit rule and the `.c.o` implicit rule, so that the dependency list for `scanner.o` extends to `scanner.l`. Since there is no rule in the target entry, `scanner.c` is built using the `.l.c` implicit rule.

The next target entry describes how to produce the `yacc` intermediate files. Because there is no implicit rule for producing both the header and the C source file using `yacc -d`, a target entry must be supplied that includes a rule for doing so.

Specifying Target Groups with the + Sign

In the target entry for `parser.c` and `parser.h`, the + sign separating the target names indicates that the entry is for a *target group*. A target group is a set of files, all of which are produced when the rule is performed. Taken as a group, the set of files comprises the target. Without the + sign, each item listed would comprise a separate target. With a target group, `make` checks the modification dates separately against each target file, but performs the target's rule only once, if necessary, per `make` run.

Maintaining Shell Scripts with `make` *and SCCS*

Although a shell script is a plain text file, it must have execute permission to run. Since SCCS removes execute permission for files under its control, it is convenient to make a distinction between a shell script and its "source" under

SCCS. make has an implicit rule for deriving a script from its source. The suffix for a shell script source file is .sh. Even though the contents of the script and the .sh file are the same, the script has execute permissions, while the .sh file does not. make's implicit rule for scripts "derives" the script from its source file, making a copy of the .sh file (retrieving it first, if necessary) and changing the mode of the resulting script file to allow execution. For example:

```
$ file script.sh
script.sh:    ascii text
$ make script
cat script.sh > script
chmod +x script
$ file script
script:       commands text
```

Running Tests with make

Shell scripts are often helpful for running tests and performing other routine tasks that are either interactive or don't require make's dependency checking. Test suites, in particular, often entail providing a program with specific, repeatable input that a program might expect to receive from a terminal.

In the case of a library, a set of programs that exercise its various functions may be written in C, and then executed in a specific order, with specific inputs from a script. In the case of a utility program, there may be a set of benchmark programs that exercise and time its functions. In each of these cases, the commands to run each test can be incorporated into a shell script for repeatability and easy maintenance.

Once you have developed a test script that suits your needs, including a target to run it is easy. Although make's dependency checking may not be needed within the script itself, you *can* use it to make sure that the program or library is updated before running those tests.

In the following target entry for running tests, `test` depends on `lib.a`. If the library is out of date, `make` rebuilds it and proceeds with the test. This insures that you always test with an up-to-date version:

```
#This is the library we're testing
LIBRARY= lib.a

test: $(LIBRARY) testscript
    set -x ; testscript > /tmp/test.\$\$

testscript: testscript.sh test_1 test_2 test_3

#rules for building the library
$(LIBRARY):
    @ echo Building $(LIBRARY)
    (library-building rules here)

#test_1 ... test_3 exercise various library functions
test_1 test_2 test_3: $$@.c $(LIBRARY)
    $(LINK.c) -o $@ $<
```

`test` also depends on `testscript`, which in turn depends on the three test programs.

This ensures that they too are up-to-date before `make` initiates the test procedure. `lib.a` is built according to its target entry in the makefile; `testscript` is built using the `.sh` implicit rule; and the test programs are built using the rule in the last target entry, assuming that there is just one source file for each test program. (The `.c` implicit rule doesn't apply to these programs because they must link with the proper libraries in addition to their `.c` files).

Escaped References to a Shell Variable

The string `\$\$` in the rule for `test` illustrates how to escape the dollar-sign from interpretation by `make`. `make` passes each `$` to the shell, which expands the `$$` to its process ID . This technique allows each test to write to a unique temporary filename. The `set -x` command forces the shell to display the commands it runs on the terminal, which allows you to see the actual file name containing the results of the specific test.

Shell Command Substitutions

You can supply shell command substitutions within a rule as in the following example:

```
do:
    @echo ‘cat Listfile‘
```

You can even place the backquoted expression in a macro:

```
DO= ‘cat Listfile‘
do:
    @echo $(DO)
```

However, you can only use this form of command substitution within a rule.

Command Replacement Macro References

If you supply a shell command as the definition of a macro:

```
COMMAND= cat Listfile
```

you can use a *command replacement macro reference* to instruct `make` to replace the reference with the output of the command in the macro’s value. This form of command substitution can occur anywhere within a makefile:

```
COMMAND= cat Listfile
$(COMMAND:sh): $$(@:=.c)
```

This example imports a list of targets from another file and indicates that each target depends on a corresponding `.c` file.

As with shell command substitution, a command replacement reference evaluates to the standard output of the command. NEWLINE characters are converted to SPACE characters. The command is performed whenever the reference is encountered. The command’s standard error is ignored. However, if the command returns a non zero exit status, `make` halts with an error.

A workaround for this is to append the `true` command to the command line:

```
COMMAND = cat Listfile ; true
```

Command Replacement Macro Assignment

A macro assignment of the form

cmd_macro`:sh = `*command*

assigns the standard output of the indicated *command* to *cmd_macro*; for instance:

```
COMMAND:sh = cat Listfile
$(COMMAND): $$(@:=.c)
```

is equivalent to the previous example. However, with the assignment form, the command is only performed once per `make` run. Again, only the standard output is used, NEWLINE characters are converted to SPACE characters, and a non zero exit status halts `make` with an error.

Alternate forms of command replacement macro assignments are:

macro`:sh += `*command*
Append command output to the value of *macro*.

target `:=` *macro*`:sh = `*command*
Conditionally define *macro* to be the output of *command* when processing *target* and its dependencies.

target `:=` *macro*`:sh += `*command*
Conditionally append the output of *command* to the value of *macro* when processing *target* and its dependencies.

Maintaining Software Projects

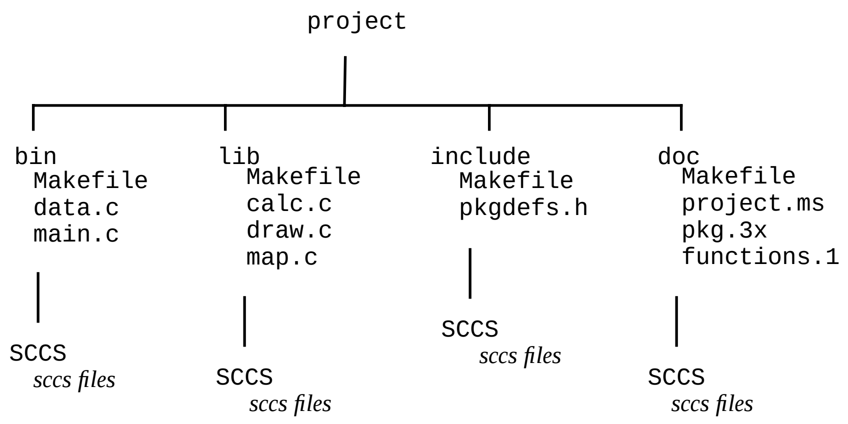
`make` is especially useful when a software project consists of a system of programs and libraries. By taking advantage of nested `make` commands, you can use it to maintain object files, executables, and libraries in a whole hierarchy of directories. You can use `make` in conjunction with SCCS to ensure that sources are maintained in a controlled manner, and that programs built from them are consistent. You can provide other programmers with duplicates of the directory hierarchy for simultaneous development and testing if you wish (although there are trade-offs to consider).

You can use `make` to build the entire project and install final copies of various modules onto another file system for integration and distribution.

Organizing a Project for Ease of Maintenance

As mentioned earlier, one good way to organize a project is to segregate each major piece into its own directory. A project broken out this way usually resides within a single filesystem or directory hierarchy. Header files could reside in one subdirectory, libraries in another, and programs in still another. Documentation, such as reference pages, may also be kept on hand in another subdirectory.

Suppose that a project is composed of one executable program, one library that you supply, a set of headers for the library routines, and some documentation, as in the following diagram.



The makefiles in each subdirectory can be borrowed from examples in earlier sections, but something more is needed to manage the project as a whole. A carefully structured makefile in the root directory, the *root makefile* for the project, provides target entries for managing the project as a single entity.

As a project grows, the need for consistent, easy-to-use makefiles also grows. Macros and target names should have the same meanings no matter which makefile you are reading. Conditional macro definitions and compilation options for output variants should be consistent across the entire project.

Where feasible, a *template* approach to writing makefiles makes sense. This makes it easy for you keep track of how the project gets built. All you have to do to add a new type of module is to make a new directory for it, copy an appropriate makefile into that directory, and make a few edits.

Of course, you also need to add the new module to the list of things to build in the root makefile.

Conventions for macro and target names, as those used in the default makefile, should be instituted and observed throughout the project. Mnemonic names mean that although you may not remember the exact function of a target or value of a macro, you'll know the type of function or value it represents (and that's usually more valuable when deciphering a makefile anyway).

Using `include` *Makefiles*

One method of simplifying makefiles, while providing a consistent compilation environment, is to use the `make`

`include` *filename*

directive to read in the contents of a named makefile; if the named file is not present, `make` checks for a file by that name in `/etc/default`.

For instance, there is no need to duplicate the pattern-matching rule for processing `troff` sources in each makefile, when you can `include` its target entry, as shown below.

```
SOURCES= doc.ms spec.ms
...
clean: $(SOURCES)
include ../pm.rules.mk
```

Here, `make` reads in the contents of the `../pm.rules.mk` file:

```
# pm.rules.mk
#
# Simple "include" makefile for pattern-matching
# rules.

%.tr: %.ms
    troff -t -ms $< > $@
%.nr: %.ms
    nroff -ms $< > $@
```

Installing Finished Programs and Libraries

When a program is ready to be released for outside testing or general use, you can use `make` to install it. Adding a new target and new macro definition to do so is easy:

```
DESTDIR= /proto/project/bin

install: functions
    -mkdir $(DESTDIR)
    cp functions $(DESTDIR)
```

A similar target entry can be used for installing a library or a set of headers.

Building the Entire Project

From time to time it is necessary to take a snapshot of the sources and the object files that they produce. Building an entire project is simply a matter of invoking `make` successively in each subdirectory to build and install each module.

The following (rather simple) example shows how to use nested `make` commands to build a simple project.

Assume your project is located in two different subdirectories, `bin` and `lib`, and that in both subdirectories you want `make` to debug, test, and install the project.

First, in the projects main, or `root`, directory, you put a makefile such as this:

```
# Root makefile for a project.

TARGETS= debug test install
SUBDIRS= bin lib

all: $(TARGETS)
$(TARGETS):
    @for i in $(SUBDIRS) ; \
    do \
       cd $$i ; \
       echo "Current directory:  $$i" ;\
       $(MAKE) $@ ; \
       cd .. ; \
    done
```

Then, in each subdirectory (in this case, `bin`) you would have a makefile of this general form:

```
#Sample makefile in subdirectory
debug:
    @echo "   Building debug target"
    @echo
test:
    @echo "   Building test target"
    @echo
install:
    @echo "   Building install target"
    @echo
```

When you type make (in the base directory), you get the following output:

```
$ make
Current directory:  bin
    Building debugging target

Current directory:  lib
    Building debugging target

Current directory:  bin
    Building testing target

Current directory:  lib
    Building testing target

Current directory:  bin
    Building install target

Current directory:  lib
    Building install target
$
```

Maintaining Directory Hierarchies with the Recursive Makefiles

If you extend your project hierarchy to include more layers, chances are that not only will the makefile in each intermediate directory have to produce target files, but it will also have to invoke nested make commands for subdirectories of its own.

Files in the current directory can sometimes depend on files in subdirectories, and their target entries need to depend on their counterparts in the subdirectories.

The nested make command for each subdirectory should run before the command in the local directory does. One way to ensure that the commands run in the proper order is to make a separate entry for the nested part and another for the local part. If you add these new targets to the dependency list for the original target, its action will encompass them both.

Maintaining Recursive Targets

Targets that encompass equivalent actions in both the local directory and in subdirectories are referred to as *recursive* targets.[1] A makefile with recursive targets is referred to as a *recursive* makefile.

In the case of `all`, the nested dependencies are `NESTED_TARGETS`; the local dependencies, `LOCAL_TARGETS`:

```
NESTED_TARGETS=  debug test install
SUBDIRS= bin lib
LOCAL_TARGETS= functions

all: $(NESTED_TARGETS) $(LOCAL_TARGETS)

$(NESTED_TARGETS):
    @ for i in $(SUBDIRS) ; \
    do \
       echo "Current directory:  $$i" ;\
       cd $$i ; \
       $(MAKE) $@ ; \
       cd .. ; \
    done

$(LOCAL_TARGETS):
    @ echo "Building $@ in local directory."
    (local directory commands)
```

The nested `make` must also be recursive, unless it is at the bottom of the hierarchy. In the makefile for a leaf directory (one with no subdirectories to go to), you only build local targets.

1. Strictly speaking, any target that calls `make`, with its name as an argument, is recursive. However, here the term is reserved for the narrower case of targets that have both nested and local actions. Targets that only have nested actions are referred to as “nested” targets.

Maintaining a Large Library as a Hierarchy of Subsidiaries

When maintaining a very large library, it is sometimes easier to break it up into smaller, subsidiary libraries, and use `make` to combine them into a complete package. Although you cannot combine libraries directly with `ar`, you can extract the member files from each subsidiary library, and then archive those files in another step, as shown in the following example.

```
$ ar xv libx.a
x - x1.o
x - x2.o
x - x3.o
$ ar xv liby.a
x - y1.o
x - y2.o
$ ar rv libz.a *.o
a - x1.o
a - x2.o
a - x3.o
a - y1.o
a - y2.o
ar: creating libz.a
```

A subsidiary library is maintained using a makefile in its own directory, along with the (object) files it is built from. The makefile for the complete library typically makes a symbolic link to each subsidiary archive, extracts their contents into a temporary subdirectory, and archives the resulting files to form the complete package.

In general, use of shell filename wild cards is considered to be bad form in a makefile. If you ***do*** use them, you need to take steps to insure that it excludes spurious files by isolating affected files in a temporary subdirectory.

The next example updates the subsidiary libraries, creates a temporary directory in which to put extracted the files, and extracts them. It uses the * (shell) wild card within that temporary directory to generate the collated list of files. While filename wild cards are generally frowned upon, this use of the wild card is acceptable because a new directory is created whenever the target is built. This guarantees that it will contain only files extracted during the *current* `make` run.

The example relies on a naming convention for directories. The name of the directory is taken from the basename of the library it contains. For instance, if `libx.a` is a subsidiary library, the directory that contains it is named `libx`.

It makes use of suffix replacements in dynamic-macro references to derive the directory name for each specific subdirectory. (You can verify that this is necessary.) It uses a shell `for` loop to successively extract each library and a shell command substitution to collate the object files into proper sequence for linking (using `lorder` and `tsort`) as it archives them into the package. Finally, it removes the temporary directory and its contents.

```
# Makefile for collating a library from subsidiaries.

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS:  libz.a

all: lib.a

libz.a: libx.a liby.a
    -rm -rf tmp
    -mkdir tmp
    set -x ; for i in libx.a liby.a ; \
       do ( cd tmp ; ar x ../$$i ) ; done
    ( cd tmp ; rm -f *_*_.SYMDEF ; ar cr ../$@ ‘lorder * | tsort‘ )
    -rm -rf tmp libx.a liby.a

libx.a liby.a: FORCE
    -cd $(@:.a=) ; $(MAKE) $@
    -ln -s $(@:.a=)/$@ $@
FORCE:
```

For the sake of clarity, this example omits support for alternate variants, as well as the targets for `clean`, `install`, and `test` (does not apply since the source files are in the subdirectories).

The `rm -f *_*_.SYMDEF` command embedded in the collating line prevents a symbol table in a subsidiary (produced by running `ar` on that library) from being archived in this library.

Since the nested `make` commands build the subsidiary libraries before the current library is processed, it is a simple matter to extend this makefile to account for libraries built from both subsidiaries and object files in the current directory. You need only add the list of object files to the dependency list for the library and a command to copy them into the temporary subdirectory for collation with object files extracted from subsidiary libraries.

```
# Makefile for collating a library from subsidiaries and local
objects.

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS:  libz.a

OBJECTS= map.o calc.o draw.o

all: libz.a

libz.a: libx.a liby.a $(OBJECTS)
    -rm -rf tmp
    -mkdir tmp
    -cp $(OBJECTS) tmp
    set -x ; for i in libx.a liby.a ; \
       do ( cd tmp ; ar x ../$$i ) ; done
    ( cd tmp ; rm -f *_*_.SYMDEF ; ar cr ../$@ ‘lorder * | tsort‘ )
       -rm -rf tmp lix.a liby.a

libx.a liby.a: FORCE
    -cd $(@:.a=) ; $(MAKE) $@
    -ln -s $(@:.a=)/$@ $@
FORCE:
```

Reporting Hidden Dependencies to `make`

You may need to write a command for processing hidden dependencies. For instance, you may need to trace document source files that are included in a `troff` document by way of `.so` requests. When `.KEEP_STATE` is in effect, `make` sets the environment variable `SUNPRO_DEPENDENCIES` to the value:

`SUNPRO_DEPENDENCIES='`*report-file target*`'`

After the command has terminated, `make` checks to see if the file has been created, and if it has, `make` reads it and writes reported dependencies to `.make.state` in the form:

target: *dependency*. . .

where *target* is the same as in the environment variable.

make *Enhancements Summary*

The following summarizes additional new features to make.

Default Makefile

make's implicit rules and macro definitions are no longer hard-coded within the program itself. They are now contained in the default makefile /usr/share/lib/make/make.rules. make reads this file automatically unless there is a file in the local directory named make.rules. When you use a local make.rules file, you must add a directive to include the standard make.rules file to get the standard implicit rules and predefined macros.

The State File .make.state

make also reads a state file, .make.state, in the directory. When the special-function target .KEEP_STATE is used in the makefile, make writes out a cumulative report for each target containing a list of hidden dependencies (as reported by compilation processors such as cpp) and the most recent rule used to build each target. The state file is very similar in format to an ordinary makefile.

Hidden-Dependency Checking

When activated by the presence of the .KEEP_STATE target, make uses information reported from cc, cpp, f77, ld, make, pc and other compilation commands and performs a dependency check against any header files (or in some cases, libraries) that are incorporated into the target file. These "hidden" dependency files do not appear in the dependency list, and often do not reside in the local directory.

Command-Dependency Checking

When .KEEP_STATE is in effect, if any command line used to build a target changes between make runs (either as a result of editing the makefile or because of a different macro expansion), the target is treated as if it were out of date; make rebuilds it (even if it is newer than the files it depends on).

Automatic Retrieval of SCCS *Files*

This section discusses the rule for the automatic retrieval of files under `sccs`.

Tilde Rules Superseded

This version of `make` automatically runs `sccs get`, as appropriate, when there is no rule to build a target file. A tilde appended to a suffix in the suffixes list indicates that `sccs` extraction is appropriate for the dependency file. `make` no longer supports tilde suffix rules that include commands to extract current versions of `sccs` files.

To inhibit or alter the procedure for automatic extraction of the current `sccs` version, redefine the `.SCCS_GET` special-function target. An empty rule for this target entirely inhibits automatic extraction.

Pattern-Matching Rules

Pattern-matching rules have been added to simplify the process of adding new implicit rules of your own design. A target entry of the form:

```
tp%ts : dp%ds
    rule
```

defines a pattern-matching rule for building a target from a related dependency file. *tp* is the target prefix; *ts*, its suffix. *dp* is the dependency prefix; *ds*, its suffix. The `%` symbol is a wild card that matches a contiguous string of zero or more characters appearing in both the target and the dependency file name. For example, the following target entry defines a pattern-matching rule for building a `troff` output file, with a name ending in `.tr` from a file that uses the `-ms` macro package ending in `.ms`:

```
%.tr: %.ms
    troff -t -ms $< > $@
```

With this entry in the makefile, the command:

```
    make doc.tr
```

produces:

```
$ make doc.tr
troff -t -ms doc.ms > doc.tr
```

Using that same entry, if there is a file named `doc2.ms`, the command:

```
make doc2.tr
```

produces:

```
$ make doc2.tr
troff -t -ms doc2.ms > doc2.tr
```

An explicit target entry overrides any pattern-matching rule that might apply to a target. Pattern-matching rules, in turn, normally override implicit rules. An exception to this is when the pattern matching rule has no commands in the rule portion of its target entry. In this case, `make` continues the search for a rule to build the target, and uses as its dependency the file that matched the (dependency) pattern.

Pattern-Replacement Macro References

As with suffix rules and pattern-matching rules, pattern replacement macro references have been added to provide a more general method for altering the values of words in a specific macro reference than that already provided by suffix replacement in macro references. A pattern-replacement macro reference takes the form:

$ (*macro* :*p* %*s* =*np* %*ns*)

where *p* is an existing prefix (if any), *s* is an existing suffix (if any), *np* and *ns* are the new prefix and suffix, and `%` is a wild card character matching a string of zero or more characters within a word.

The prefix and suffix replacements are applied to all words in the macro value that match the existing pattern. Among other things, this feature is useful for prefixing the name of a subdirectory to each item in a list of files. For instance, the following makefile:

```
SOURCES= x.c y.c z.c
SUBFILES.o= $(SOURCES:%.c=subdir/%.o)

all:
    @echo $(SUBFILES.o)
```

produces:

```
$ make
subdir/x.o subdir/y.o subdir/z.o
```

You may use any number of % wild cards in the right-hand (replacement) side of the = sign, as needed. The following replacement:

```
...
NEW_OBJS= $(SOURCES:%.c=%/%.o)
```

would produce:

```
...
x/x.o y/y.o z/z.o
```

Please note that pattern-replacement macro references should not appear on the dependency line of a pattern-matching rule's target entry. This produces unexpected results. With the makefile:

```
OBJECT= .o

x:
%: %.$(OBJECT:%o=%Z)
    cp $< $@
```

it looks as if `make` should attempt to build a target named `x` from a file named `x.Z`. However, the pattern-matching rule is not recognized; `make` cannot determine which of the % characters in the dependency line apply to the pattern-matching rule and that apply to the macro reference.

Consequently, the target entry for `x.Z` is never reached. To avoid problems like this, you can use an intermediate macro on another line:

```
OBJECT= .o
ZMAC= $(OBJECT:%o=%Z)

x:
%: %$(ZMAC)
    cp $< $@
```

New Options

The new options are:

–d

Display dependency-check results for each target processed. Displays all dependencies that are newer, or indicates that the target was built as the result of a command dependency.

–dd

The same function as `-d` had in earlier versions of `make`. Displays a great deal of output about all details of the `make` run, including internal states, etc.

`-D`

Display the text of the makefile as it is read.

`-DD`

Display the text of the makefile and of the default makefile being used.

`-p`

Print macro definitions and target entries.

`-P`

Report all dependencies for targets without rebuilding them.

Support for C++ and Modula-2

This version of `make` contains predefined macros for compiling C++ programs. It also contains predefined macros and implicit rules for compiling Modula-2.

Naming Scheme for Predefined Macros

The naming scheme for predefined macros has been rationalized, and the implicit rules have been rewritten to reflect the new scheme. The macros and implicit rules are upward compatible with existing makefiles.

Some examples include the macros for standard compilations commands:

`LINK.c`

Standard `cc` command line for producing executable files.

`COMPILE.c`

Standard `cc` command line for producing object files.

New Special-Purpose Targets

`.KEEP_STATE`

The .KEEP_STATE target should not be removed once it has been used in a `make` run.

When included in a makefile, this target enables hidden dependency and command-dependency checking. In addition, `make` updates the state file `.make.state` after each run.

`.INIT and .DONE`

These targets can be used to supply commands to perform at the beginning and end of each `make` run.

`.FAILED`

The commands supplied are performed when `make` fails.

`.PARALLEL`

These can be used to indicate which targets are to be processed in parallel, and which are to be processed in serial fashion.

`.SCCS_GET`

This target contains the rule for extracting current versions of files from `sccs` history files.

`.WAIT`

When this target appears in the dependency list, `make` waits until the dependencies that precede it are finished before processing those that follow, even when processing is parallel.

New Implicit `lint` *Rule*

Implicit rules have been added to support incremental verification with `lint`.

Macro Processing Changes

A macro value can now be of virtually any length. Whereas in earlier versions only trailing white space was stripped from a macro value, this version strips off both leading and trailing white space characters.

Macros: Definition, Substitution, and Suffix Replacement

New Append Operator

`+=`

This is the new append operator that appends a SPACE followed by a word or words, onto the existing value of the macro.

Conditional Macro Definitions

`:=`

This is the conditional macro definitions operator that indicates a conditional (targetwise) macro definition. A makefile entry of the form:

target `:=` *macro* `=` *value*

indicates that *macro* takes the indicated *value* while processing *target* and its dependencies.

Patterns in Conditional Macros

`make` recognizes the `%` wild card pattern in the target portion of a conditional macro definition. For instance:

```
profile_% := CFLAGS += -pg
```

would modify the `CFLAGS` macro for all targets having the ‘`profile_`’ prefix. Pattern replacements can be used within the value of a conditional definition. For instance:

```
profile_% := OBJECTS = $(SOURCES:%.c=profile_%.o)
```

applies the `profile_` prefix and `.o` suffix to the basename of every `.c` file in the `SOURCES` list (value).

Suffix Replacement Precedence

Substring replacement now takes place following expansion of the macro being referred to. Previous versions of `make` applied the substitution first, with results that were counterintuitive.

Nested Macro References

`make` now expands inner references before parsing the outer reference. A nested reference as in this example:

```
CFLAGS-g = -I../include
OPTION = -g
$(CFLAGS$(OPTION))
```

now yields the value `-I../include`, rather than a null value, as it would have in previous versions.

Cross-Compilation Macros

The predefined macros `HOST_ARCH` and `TARGET_ARCH` are available for use in cross-compilations. By default, the *arch* macros are set to the value returned by the `arch` command..

Shell Command Output in Macros

A definition of the form:

MACRO `:sh =` *command*

sets the value of *MACRO* to the standard output of the indicated *command*, NEWLINE characters being replaced with SPACE characters. The command is performed just once, when the definition is read. Standard error output is ignored, and `make` halts with an error if the command returns a non zero exit status.

A macro reference of the form:

`$(`*MACRO* `:sh)`

expands to the output of the command line stored in the value of *MACRO*, whenever the reference is evaluated. NEWLINE characters are replaced with SPACE characters, standard error output is ignored, and `make` halts with an error if the command returns a non zero exit status.

Improved `ar` *Library Support*

`make` automatically updates an `ar`-format library member from a file having the same name as the member. Also, `make` now supports lists of members as dependency names of the form:

lib.a`:` *lib.a*`(`*member member . . .* `)`

Target Groups

It is now possible to specify that a rule produces a set of target files. A + sign between target names in the target entry indicates that the named targets comprise a group. The target group rule is performed once, at most, in a make invocation.

Incompatibilities with Previous Versions

This section briefly discusses the following:

- The -d Option
- Dynamic Macros
- Tilde Rules
- Target Names

The -d Option

The –d option now reports the reason why a target is considered out of date.

Dynamic Macros

Although the dynamic macros $< and $* were documented as being assigned only for implicit rules and the .DEFAULT target, in some cases they actually were assigned for explicit target entries. The assignment action is now documented properly.

The actual value assigned to each of these macros is derived by the same procedure used within implicit rules (this hasn't changed). This can lead to unexpected results when they are used in explicit target entries.

Even if you supply explicit dependencies, make doesn't use them to derive values for these macros. Instead, it searches for an appropriate implicit rule and dependency file. For instance, if you have the explicit target entry:

```
test: test.f
    @echo $<
```

and the files: test.c and test.f, you might expect that $< would be assigned the value test.f. This is *not* the case. It is assigned **test.c**, because .c is ahead of .f in the suffixes list:

```
$ make test
test.c
```

For explicit entries, it is best to use a strictly deterministic method for deriving a dependency name using macro references and suffix replacements. For example, you could use $@.f instead of $< to derive the dependency name. To derive the base name of a .o target file, you could use the suffix replacement macro reference: $(@:.o=) instead of $*.

When hidden dependency checking is in effect, the $? dynamic macro value includes the names of hidden dependencies, such as header files. This can lead to failed compilations when using a target entry such as:

```
x: x.c
    $(LINK.c) -o $@ $?
```

and the file x.c #include's header files. The workaround is to replace '$?' with '$@.<'.

Tilde Rules

Tilde rules are not supported. This version of make does not support tilde suffix rules for version retrieval under SCCS. This may create problems when older makefiles redefine tilde rules to perform special steps when version retrieval under SCCS is required.

Target Names

Target names beginning with ./ are treated as local filenames.

When make encounters a target name beginning with './', it strips those leading characters. For instance, the target named:

```
./filename
```

is interpreted as if it were written:

```
filename
```

This can result in endless loop conditions when used in a recursive target. To avoid this, rewrite the target relative to ‘`..`’, the parent directory:

```
../dir/filename
```