Make

Nowadays, only the most trivial UNIX package comes without a *Makefile*, and you can assume that the central part of building just about any package is:

```
$ make
```

We won't go into the details of how *make* works here—you can find this information in *Managing projects with make*, by Andrew Oram and Steve Talbott. In this chapter, we'll look at the aspects of *make* that differ between implementations. We'll also take a deeper look at BSD *make*, because it is significantly different from other flavours, and because there is very little documentation available for it.

Terminology

In the course of evolution of *make*, a change in terminology has taken place. Both the old and the new terminology are in current use, which can be confusing at times. In the following list, we'll look at the terms we use in this book, and then relate them to others which you might encounter:

- A *rule* looks like:

```
target:  dependencies
         command
         command
```

- A *target* is the name by which you invoke a rule. *make* implicitly assumes that you want to create a file of this name.
- The *dependencies* are the files or targets upon which the target *depends*: if any of the dependencies do not exist, or they are newer than the current target file, or the corresponding target needs to be rebuild, then the target will be rebuilt (in other words, its commands will be executed). Some versions of *make* use the terms *prerequisite* or *source* to represent what we call dependencies.
- The *commands* are single-line shell scripts that get executed in sequence if the target needs to be rebuilt.

- *variables* are environment variables that *make* imports, explicitly named variables, or *implicit variables* such as `$@` and `$<`. Variables used to be called *macro*s. They aren't really macros, since they don't take parameters, so the term *variable* is preferable. BSD *make* uses the term *local variable* for implicit variables. As we will see, they don't correspond exactly. SunOS uses the term *dynamic macros* for implicit variables.

Additional make features

A number of versions of *make* offer additional features beyond those of the version of *make* described in *Managing projects with make*. In the following sections, we'll look at:

- Internal variables
- Variables with special meanings
- Targets with special meanings
- Including other source files from the *Makefile*
- Conditional execution
- Variations on assignments to variables
- Functions
- Multiple targets

Internal variables

All versions of *make* supply internal variables, but the list differs between individual implementations. We'll defer their discussion until we discuss BSD *make*, on page 324.

Variables with special meanings

A number of normal variables have taken on special meanings in some versions of *make*. Here's an overview:

- `VPATH` is a list of directory names to search for files named in dependencies. It is explicitly supported in GNU *make*, where it applies to all file searches, and is also supported, but not documented, in some versions of System V.4. GNU *make* also supports a directive `vpath`.
- `MAKE` is the name with which *make* was invoked. It can be used to invoke subordinate *make*s, and has the special property that it will be invoked even if you have specified the `-n` flag to *make*, indicating that you just want to see the commands that would be executed, and you don't want to execute them.
- In all modern versions of *make*, `MAKEFLAGS` is a list of the flags passed to *make*. *make* takes the value of the environment variable `MAKEFLAGS`, if it exists, and adds the command line arguments to it. It is automatically passed to subordinate *make*s.

- `SHELL` is the name of a shell to be used to execute commands. Note that many versions of *make* execute simple commands directly, so you may find that this doesn't have any effect unless you include a shell metacharacter like `;`.

The exact semantics of these variables frequently varies from one platform to another—in case of doubt, read your system documentation.

Special targets

All versions of *make* define a number of targets that have special meanings. Some versions define additional targets:

- `.BEGIN` is a target to be executed before any other target. It is supported by BSD *make*.
- `.INIT` is a target to be executed before any other target. It is supported by SunOS and Solaris *make*.
- `.END` is a target to be executed after all other targets have been executed. It is supported by BSD *make*.
- `.DONE` is a target to be executed after all other targets have been executed. It is supported by SunOS and Solaris *make*.
- `.FAILED` is a target to be executed after all other targets have been executed. It is supported by SunOS and Solaris *make*.
- `.INTERRUPT` is a target to be executed if *make* is interrupted. It is supported by BSD *make*.
- `.MAIN` is the default target to be executed if no target was specified on the command line. If this target is missing, *make* will execute the first target in the *Makefile*. It is supported by BSD *make*.
- `.MAKEFLAGS` is an alternate method to supply flags to subordinate *make*s. It is supported by BSD *make*.
- `.PATH` is an alternate method to specify a search path for files not found in the current directory. It is supported by BSD *make*.
- `.MUTEX` is used in System V.4 to synchronize parallel *make*s.
- GNU *make* uses the target `.PHONY` to indicate targets that do not create files, such as `clean` and `install`. If by chance you have a file *install* in your directory, *make* will determine that `make install` does not need to be executed, since *install* is up to date. If you use GNU *make*, you can avoid this problem with:

  ```
  .PHONY: all install clean
  ```

  If you don't have GNU *make*, you can usually solve the problem with

```
all install clean:       .FORCE
        install commands

.FORCE:
```

In this example, `.FORCE` looks like a special target, as it is meant to. In fact, the name is not important: you just need a name that doesn't correspond to a real file.

In addition to special targets, BSD *make* also has *special sources* (in other words, special dependencies). We'll look at them on page 327.

include directive

Many modern *make*s allow you to include other files when processing the *Makefile*. Unfortunately, the syntax is very variable:

- In GNU *make*, the syntax is simply `include` *`filename`*.
- In BSD *make*, the syntax is `.include <`*`filename`*`>` or `.include "`*`filename`*`"`. The syntax resembles that of the C preprocessor: the first form searches only the system directories, the second form searches the current directory before searching the system directories.
- In SunOS, Solaris and System V.4 *make*, the syntax is `include` *`filename`*, but the text `include` must be at the beginning of the line.
- SunOS and Solaris *make* automatically include a file *make.rules* in the current directory if it exists. Otherwise they include the file */usr/share/lib/make/make.rules*.

Conditional execution

A number of versions of *make* support conditional execution of commands. GNU *make* has commands reminiscent of the C preprocessor:

```
ifeq (${CC},gcc}
        ${CC} -traditional -O3 -g $*.c -c -o $<
else
        ${CC} -O $*.c -c -o $<
endif
```

BSD *make* has a different syntax, which also vaguely resembles the C preprocessor. Apart from standard `.if`, `.else` and `.endif`, BSD *make* also provides an `.ifdef` directive and additional operators analogous to `#if defined`:

- `.if make (`*`variable`*`)` checks whether *variable* is a main target of *make* (in other words, if it was mentioned on the command line that invoked *make*).
- `.if empty (`*`variable`*`)` tests whether *variable* represents the empty string.
- `.if exists (`*`variable`*`)` tests whether the file *variable* exists.

- `.if target (`*`variable`*`)` tests whether *variable* represents a defined target.

SunOS and Solaris have so-called *conditional macros*:

```
foo bar baz:=    CC = mycc
```

This tells *make* that the variable (macro) `CC` should be set to `mycc` only when executing the targets `foo`, `bar`, and `baz`.

Other forms of variable assignment

Simply expanded variables

make normally expands variables until no variable references remain in the result. Take the following *Makefile*, for example:

```
CFLAGS = $(INCLUDE) $(OPT)
OPT    = -g -O3
INCLUDE= -I/usr/monkey -I/usr/dbmalloc

all:
        @echo CFLAGS: ${CFLAGS}
```

If you run *make*, you will get:

```
$ make
CFLAGS: -I/usr/monkey -I/usr/dbmalloc -g -O3
```

On the other hand, you can't change the definition to:

```
CFLAGS = $(CFLAGS) -I/usr/monkey
```

If you do this, you will get:

```
$ make
makefile:7: *** Recursive variable ‘CFLAGS’ references itself (eventually).  Stop.
```

make would loop trying to expand `$(CFLAGS)`. GNU *make* solves this with *simply expanded* variables, which go through one round of expansion only. You specify them with the assignment operator `:=` instead of the usual `=`. For example:

```
CFLAGS = -g -O3
CFLAGS := $(CFLAGS) -I/usr/monkey
```

In this case, `CFLAGS` expands to `-g -O3 -I/usr/monkey`.

define directive

You frequently see multi-line shell script fragments in *make* rules. They're ugly and error-prone, because in conventional *make*, you need to put this command sequence on a single line with lots of backslashes and semicolons. GNU *make* offers an alternative with the *define* directive. For example, to check for the existence of a directory and create it if it doesn't exist, you might normally write

```
${INSTDIR}:
        if [ ! -d $@ ]; then \
          mkdir -p $@; \
        fi
```

With GNU *make*, you can define this as a command:

```
define makedir
if [ ! -d $@ ]; then
  mkdir -p $@
fi
endef

${INSTDIR}:
        ${makedir}
```

Override variable definitions

Conventional versions of *make* have three ways to define a *make* variable. In order of precedence, they are:

1. Define it on the command line used to invoke *make*:

   ```
   $ make CFLAGS="-g -O3"
   ```

2. Define it in the *Makefile*.

3. Define it in an environment variable. This is all the more confusing because most shells allow you to write the environment variable on the same line as the invokation of *make*:

   ```
   $ CFLAGS="-g -O3" make
   ```

   This looks almost identical to the first form, but the precedence is lower.

The command line option has the highest priority. This is usually a good idea, but there are times when you want the declaration in the *Makefile* to take precedence: you want to override the definition on the command line. GNU *make* allows you to specify it with the *override* directive. For example, you might want to insist that the optimization level be limited to `-O2` if you're generating debugging symbols. In GNU *make*, you can write:

```
override CFLAGS=-O2
```

Functions

As well as variables, GNU *make* supplies builtin *functions*. You call them with the syntax `${function arg,arg,arg}`. These functions are intended for text manipulation and have names like `subst`, `findstring`, `sort`, and such. Unfortunately there is no provision for defining your own functions.

Multiple targets

All forms of *make* support the concept of *multiple targets*. They come in two flavours:

- *Single-colon targets*, where the target name is followed by a single colon. Each target of the same name may specify dependencies—this is how *Makefile* dependencies are specified—but only one rule may have commands. If any of the dependencies require the target to be rebuilt, then these commands will be executed. If you supply commands to more than one rule, the behaviour varies: some versions of *make* will print a warning or an error message, and generally they execute only the last rule with commands. Under these circumstances, however, BSD *make* executes the first rule with commands.
- *Double-colon targets* have two colons after the target name. Each of these is independent of the others: each may contain commands, and each gets executed only if the dependencies for that rule require it. Unfortunately, if multiple rules need to be executed, the sequence of execution of the rules is not defined. Most versions of *make* execute them in the sequence in which they appear in the *Makefile*, but it has been reported that some versions of BSD *make* execute in reverse order, which breaks some *Imakefiles*.

BSD make

With the Net/2 release, the Computer Sciences Research Group in Berkeley released a completely new *make* with many novel features. Most BSD flavoured software that has come out in the last few years uses it. Unfortunately, it contains a number of incompatibilities with other makes. It is part of the 4.4BSD Lite distribution—see Appendix E, *Where to get sources* for further details—and includes hardcopy documentation, which refers to it as *PMake*. This name does not occur anywhere else, though you may see the name *bsdmake*.

We've already seen some of the smaller differences between BSD *make* and other flavours. In the following sections we'll look at some more significant differences. On page 327 we'll investigate the features of BSD *make* designed to make configuration easier.

Additional rule delimiter

There is a third delimiter between target and dependency in rules. Apart from the single and double colon, which have the same meaning as they do with other *makes*, there is a `!` delimiter. This is the same as the single colon delimiter in that the dependencies are the sum of all dependencies for the target, and that only the first rule set gets executed. However, the commands are always executed, even if all the dependencies are older than the target.

Assignment operators

BSD *make* supplies five different types of variable assignment:

- `=` functions as in other versions of *make*: the assignment `CFLAGS = -g` unconditionally sets `CFLAGS` to `-g`.

- += adds to a definition. If CFLAGS was set as in the previous example, writing CFLAGS += -O3 results in a new value -g -O3.
- ?= assigns a value only if the variable is currently undefined. This can be used to set default values.
- := assigns and expands immediately. This is the same as the GNU *make* := assignment.
- != expands the value and passes it to a shell for execution. The result from the shell is assigned to the variable after changing newline characters to spaces.

Variables

BSD *make* has clarified the definitions of variables somewhat. Although there is nothing really new in this area, the terminology is arranged in a more understandable manner. It defines four different kinds of variables, the first three of which correspond to the kinds of variable assignment in other *make*s. In order of priority, they are:

- Environment variables
- *global* variables (just called variables in other flavours of *make*)
- command line variables
- *local variables*, which correspond roughly to implicit variables in other *makes*.

BSD *make* allows the use of the implicit variable symbols ($@ and friends), but doesn't recommend it. They don't match very well, anyway, so it makes sense not to use them. Local variables are really variables that *make* predefines. Table 19-1 compares them to traditional *make* variables:

Table 19−1: make local variables

Traditional	BSD	Meaning
	.ALLSRC, $>	The list of all dependencies ("sources") for this target.
$ˆ		(GNU *make*) The list of all dependencies of the current target. Only the member name is returned for dependencies that represent an archive member. Otherwise this is the same as BSD .ALLSRC.
$@	.ARCHIVE	The name of the current target. If the target is an archive file member, the name of the archive file.
$$@	.TARGET, $@	The complete name of the current target, even if it represents an archive file.[1]

Table 19−1: *make* local variables (continued)

Traditional	BSD	Meaning
	`.IMPSRC, $<`	The *implied source*, in other words the name of the source file (dependency) implied in an implicit rule.
`$<`		The name of the current dependency that has been modified more recently than the target. Traditionally, it can only be used in suffix rules and in the `.DEFAULT` entry, but most modern versions of *make* (except BSD *make*) allow it to be used in normal rules as well.
`$%`	`.MEMBER`	The name of an archive member. For example, if the target name is `lib`*foo.a(bar.o)*, `$@` evaluates to `lib`*foo.a* and `$%` evaluates to `bar.o`. Supported by GNU, SunOS and System V.4 *make*.
`$?`	`.OODATE, $?`	The dependencies for this target that were newer than the target.[2]
`$*`		The raw name of the current dependency, without suffix, but possibly including directory components. Can only be used in suffix rules.
`${*F}`	`.PREFIX, $*`	The raw file name of the current dependency. It does not contain any directory component.
`${*D}`		The directory name of the current dependency. For example, if `$@` evaluates to `foo/bar.o`, `${@D}` will evaluate to `foo`. Supported by GNU, SunOS and System V.4 *make*.
	`.CURDIR`	The name of the directory in which the top-level *make* was started.

[1] `$$@` can only be used to the right of the colon in a dependency line. Supported by SunOS and System V.4 *make*.

[2] Confusingly, BSD *make* refers to these dependencies as *out of date*, thus the name of the variable.

Variable substitution

In BSD *make*, variable substitution has reached a new level of complexity. All versions of *make* support the syntax `${SRC:.c=.o}`, which replaces a list of names of the form `foo.c bar.c baz.c` with `foo.o bar.o baz.o.`. BSD *make* generalizes this syntax is into `${variable[:modifier[: . . . ]]}`. In the following discussion, BSD *make* uses the term *word* where we would normally use the term *parameter*. In particular, a file name is a word. *modifier* is an upper case letter:

- `E` replaces each word in the variable with its suffix.
- According to the documentation, `H` strips the "last component" from each "word" in the variable. A better definition is: it returns the directory name of each file name. If the original file name didn't have a directory name, the result is set to `.` (current directory).
- `M`*`pattern`* selects those words from the variable that match *pattern*. *pattern* is a *globbing pattern* such as is used by shells to specify wild-card file names.
- `N`*`pattern`* selects those words from the variable that *don't* match *pattern*.
- `R` replaces each word in the variable with everything but its suffix.
- `S`*`/old/new/`* replaces the first occurrence of the text *`old`* with *`new`*. The form `S`*`/old/new/g`* replaces all occurrences.
- `T` replaces each word in the variable with its "last component", in other words with the file name part.

This is heavy going, and it's already more than the documentation tells you. The following example shows a number of the features:

```
SRCS = foo.c bar.c baz.cc zot.pas glarp.f src/mumble.c util/grunt.f
LANGS = ${SRCS:E}
DIRS = ${SRCS:H}
OBJS = ${SRCS:T}
CSRCS = ${SRCS:M*.c}
PASSRCS = ${SRCS:M*.pas}
FSRCS = ${SRCS:M*.f}
PROGS = ${SRCS:R}
PROFS  = ${CSRCS:S/./_p./g:.c=.o}

all:
        @echo Languages: ${LANGS}
        @echo Objects: ${OBJS}
        @echo Directories: ${DIRS}
        @echo C sources: ${CSRCS}
        @echo Pascal sources: ${PASSRCS}
        @echo Fortran sources: ${FSRCS}
        @echo Programs: ${PROGS}
        @echo Profiled objects: ${PROFS}
```

If you run it, you get:

```
$ make
Languages: c c cc pas f c f
Objects: foo.c bar.c baz.cc zot.pas glarp.f mumble.c grunt.f
Directories: . . . . . src util
C sources: foo.c bar.c src/mumble.c
Pascal sources: zot.pas
Fortran sources: glarp.f util/grunt.f
Programs: foo bar baz zot glarp src/mumble util/grunt
Profiled objects: foo_p.o bar_p.o src/mumble_p.o
```

Special sources

In addition to special targets, BSD *make* includes *special sources* (recall that *source* is the word that it uses for dependencies). Here are the more important special sources:

- `.IGNORE`, `.SILENT` and `.PRECIOUS` have the same meaning as the corresponding special targets in other versions of *make*.
- `.MAKE` causes the associated dependencies to be executed even if the flags `-n` (just list commands, don't perform them) or `-t` (just update timestamps, don't perform *make*) are specified. This enables *make* to perform subsidiary *make*s even if these flags are specified. If this seems a strange thing to want to do, consider that the result of the main *make* could depend on subsidiary *make*s to such an extent that it would not even make sense to run *make -n* if the subsidiary *make*s did not run correctly—for example, if the subsidiary *make* were a *make depend*.
- `.OPTIONAL` tells *make* that the specified dependencies are not crucial to the success of the build, and that *make* should assume success if it can't figure out how to build the target.

Specifying dependencies

We have seen that the bulk of a well-written *Makefile* can consist of dependencies. BSD *make* offers the alternative of storing these files in a separate file called *.depend*. This avoids the problem of different flavours of *makedepend* missing the start of the dependencies and adding them again.

BSD Makefile configuration system

One of the intentions of BSD *make* is to make configuration easier. A good example of how much difference it makes is in the *Makefile*s for *gcc*. In its entirety, the top-level *Makefile* is:

```
SUBDIR= cc cpp lib cc1 libgcc cc1plus cc1obj #libobjc
.include <bsd.subdir.mk>
```

The complete *Makefile* in the subdirectory *cc1* (the main pass of the compiler) reads

```
#       @(#)Makefile    6.2 (Berkeley) 2/2/91

PROG=    gcc1
BINDIR=  /usr/libexec
SRCS=    c-parse.c c-lang.c  c-lex.c c-pragma.c  \
         c-decl.c c-typeck.c c-convert.c c-aux-info.c \
         c-iterate.c

CFLAGS+= -I. -I$(.CURDIR) -I$(.CURDIR)/../lib
YFLAGS=
NOMAN=   noman

.if exists(${.CURDIR}/../lib/obj)
```

```
LDADD=  -L${.CURDIR}/../lib/obj -lgcc2
DPADD=  ${.CURDIR}/../lib/obj/libgcc2.a
.else
LDADD=  -L${.CURDIR}/../lib/ -lgcc2
DPADD=  ${.CURDIR}/../lib/libgcc2.a
.endif

LDADD+= -lgnumalloc
DPADD+= ${LIBGNUMALLOC}

.include <bsd.prog.mk>
```

The standard release Makefile for *gcc* is about 2500 lines long. Clearly a lot of work has gone into getting the BSD Makefiles so small. The clue is the last line of each Makefile:

```
.include <bsd.subdir.mk>
```

or

```
.include <bsd.prog.mk>
```

These files are supplied with the system and define the hardware and software used on the system. They are normally located in */usr/share/mk*, and you can modify them to suit your local preferences.

This configuration mechanism has little connection with the new BSD *make*. It could equally well have been done, for example, with GNU *make* or System V *make*. Unfortunately, the significant incompatibilities between BSD *make* and the others mean that you can't just take the configuration files and use them with other flavours of *make*.

The BSD system places some constraints on the *Makefile* structure. To get the best out of it, you may need to completely restructure your source tree. To quote *bsd.README*:

> *It's fairly difficult to make the BSD .mk files work when you're building multiple programs in a single directory. It's a lot easier [to] split up the programs than to deal with the problem. Most of the agony comes from making the "obj" directory stuff work right, not because we switch to a new version of make. So, don't get mad at us, figure out a better way to handle multiple architectures so we can quit using the symbolic link stuff.*

On the other hand, it's remarkably easy to use BSD *make* configuration once you get used to it. It's a pity that the *make* itself is so incompatible with other *make*s: although the system is good and works well, it's usually not worth restructuring your trees and rewriting your *Makefiles* to take advantage of it.

There are a couple of other points to note about the configuration method:

- *make depend* is supported via an auxiliary file *.depend*, which *make* reads after reading the *Makefile*.
- The configuration files are included at the *end* of the *Makefile*. This is due to the way that BSD *make* works: unlike other *make*s, if multiple targets with a single colon exist, only the first will be executed, but if multiple declarations of the same variable exist, only the last one will take effect.

The configuration files consist of one file, *sys.mk*, which *make* automatically reads before doing anything else, and a number of others, one of which is usually included as the last line in a *Makefile*. These are usually:

- *bsd.prog.mk* for a *Makefile* to make an executable binary.
- *bsd.lib.mk* for a *Makefile* to make a library.
- *bsd.subdir.mk* to make binaries or libraries in subdirectories of the current directory.
- In addition, another file *bsd.doc.mk* is supplied to make hardcopy documentation. In keeping with the Cinderella nature of such parts of a package, no other file refers to it. If you want to use it, you include it *in addition* to one of the other three. This is required only for hardcopy documentation, not for *man* pages, which *are* installed by the other targets.

sys.mk

sys.mk contains global definitions for all makes. *make* reads it in before looking for any *Makefiles*. The documentation states that it is not intended to be modified, but since it contains default names for all tools, as well as default rules for makes, there is every reason to believe that you *will* want to change this file: there's no provision to override these definitions anywhere else. How you handle this dilemma is your choice. I personally prefer to change *sys.mk* (and put up with having to update it when a new release comes), but you could create another file *bsd.own.mk*, like FreeBSD does, and put your personal choices in there. The last line of the FreeBSD *sys.mk* is

```
.include <bsd.own.mk>
```

With this method you can override the definitions in *sys.mk* with the definitions in *bsd.own.mk*. It's up to you to decide whether this is a better solution.

bsd.prog.mk

bsd.prog.mk contains definitions for building programs. Table 19-2 lists the targets that it defines:

Table 19−2: bsd.prog.mk targets

Target	Purpose
`all`	Build the single program `${PROG}`, which is defined in the *Makefile*.
`clean`	remove `${PROG}`, any object files and the files *a.out*, *Errs*, *errs*, *mklog*, and *core*.
`cleandir`	remove all of the files removed by the target `clean` and also the files *.depend*, *tags*, *obj*, and any manual pages.
`depend`	make the dependencies for the source files, and store them in the file *.depend*.

Table 19−2: bsd.prog.mk targets (continued)

Target	Purpose
`install`	install the program and its manual pages. If the *Makefile* does not itself define the target `install`, the targets `beforeinstall` and `afterinstall` may also be used to cause actions immediately before and after the install target is executed.
`lint`	run lint on the source files.
`tags`	create a *tags* file for the source files.

In addition, it supplies default definitions for the variables listed in Table 19-3. The operator `?=` is used to ensure that they are not redefined if they are already defined in the *Makefile* (see page 324 for more details of the `?=` operator).

Table 19−3: variables defined in bsd.prog.mk

Variable	Purpose
`BINGRP`	Group ownership for binaries. Defaults to *bin*.
`BINOWN`	Owner for binaries. Defaults to *bin*.
`BINMODE`	Permissions for binaries. Defaults to 555 (read and execute permission for everybody).
`CLEANFILES`	Additional files that the `clean` and `cleandir` targets should remove. *bsd.prog.mk* does not define this variable, but it adds the file *strings* to the list if the variable `SHAREDSTRINGS` is defined.
`DPADD`	Additional library dependencies for the target `${PROG}`. For example, if you write `DPADD=${LIBCOMPAT} ${LIBUTIL}` in your *Makefile*, the target depends on the compatibility and utility libraries.
`DPSRCS`	Dependent sources—a list of source files that must exist before compiling the program source files. Usually for a building a configuration file that is required by all sources. Not all systems define this variable.
`LIBC`	The C library. Defaults to */lib/libc.a.*
`LIBCOMPAT`	The 4.3BSD compatibility library. Defaults to */usr/lib/libcompat.a.*
`LIBCURSES`	The *curses* library. Defaults to */usr/lib/libcurses.a.*
`LIBCRYPT`	The *crypt* library. Defaults to */usr/lib/libcrypt.a.*
`LIBDBM`	The *dbm* library. Defaults to */usr/lib/libdbm.a.*
`LIBDES`	The *des* library. Defaults to */usr/lib/libdes.a.*
`LIBL`	The *lex* library. Defaults to */usr/lib/libl.a.*

Table 19−3: variables defined in bsd.prog.mk (continued)

Variable	Purpose
`LIBKDB`	Defaults to */usr/lib/libkdb.a*.
`LIBKRB`	Defaults to */usr/lib/libkrb.a*.
`LIBM`	The math library. Defaults to */usr/lib/libm.a*.
`LIBMP`	Defaults to */usr/lib/libmp.a*.
`LIBPC`	Defaults to */usr/lib/libpc.a*.
`LIBPLOT`	Defaults to */usr/lib/libplot.a*.
`LIBTELNET`	Defaults to */usr/lib/libtelnet.a*.
`LIBTERM`	Defaults to */usr/lib/libterm.a*.
`LIBUTIL`	Defaults to */usr/lib/libutil.a*.
`SRCS`	List of source files to build the program. Defaults to `${PROG}.c`.
`STRIP`	If defined, this should be the flag passed to the install program to cause the binary to be stripped. It defaults to `-s`.

The variables in Table 19-4 are not defined in *bsd.prog.mk*, but will be used if they have been defined elsewhere:

Table 19−4: variables used by bsd.prog.mk

Variable	Purpose
`COPTS`	Additional flags to supply to the compiler when compiling C object files.
`HIDEGAME`	If defined, the binary is installed in */usr/games/hide*, and a symbolic link is created to */usr/games/dm*.
`LDADD`	Additional loader objects. Usually used for libraries.
`LDFLAGS`	Additional loader flags.
`LINKS`	A list of pairs of file names to be linked together. For example `LINKS= ${DESTDIR}/bin/test ${DESTDIR}/bin/[` links */bin/test* to */bin/[*.
`NOMAN`	If set, *make* does not try to install man pages. This variable is defined only in *bsd.prog.mk*, and not in *bsd.lib.mk* or *bsd.man.mk*.
`PROG`	The name of the program to build. If not supplied, nothing is built.
`SRCS`	List of source files to build the program. If `SRC` is not defined, it's assumed to be `${PROG}.c`.

Table 19−4: variables used by bsd.prog.mk (continued)

Variable	Purpose
SHAREDSTRINGS	If defined, the *Makefile* defines a new `.c.o` rule that uses *xstr* to create shared strings.
SUBDIR	A list of subdirectories that should be built as well as the targets in the main directory. Each target in the main Makefile executes the same target in the subdirectories. Note that the name in this file is `SUBDIR`, though it has the same function as the variable `SUBDIRS` in *bsd.subdir.mk*.

There are a couple more points to note:

- If the file *../Makefile.inc* exists, it is included before the other definitions. This is one possibility for specifying site preferences, but of course it makes assumptions about the source tree structure, so it's not completely general.
- The file *bsd.man.mk* is included unless the variable `NOMAN` is defined. We'll take another look at *bsd.man.mk* on page 333.

bsd.lib.mk

bsd.lib.mk contains definitions for making library files. It supplies the same targets as *bsd.prog.mk*, but defines or uses a much more limited number of variables:

Table 19−5: Variables defined or used in bsd.lib.mk

Variable	Purpose
LDADD	Additional loader objects.
LIB	The name of the library to build. The name is in the same form that you find in the `-l` option to the C compiler—if you want to build *libfoo.a*, you set `LIB` to `foo`.
LIBDIR	Target installation directory for libraries. Defaults to */usr/lib*.
LIBGRP	Library group owner. Defaults to *bin*.
LIBOWN	Library owner. Defaults to *bin*.
LIBMODE	Library mode. Defaults to 444 (read access for everybody).
LINTLIBDIR	Target directory for lint libraries. Defaults to */usr/libdata/lint*.
NOPROFILE	If set, only standard libraries are built. Otherwise (the default), both standard libraries (*libfoo.a*) and profiling libraries (*libfoo_p.a*) are built.*
SRCS	List of source files to build the library. Unlike in *bsd.prog.mk*, there is no default value.

Given the choice of compiling *foo.s* or *foo.c, bsd.lib.mk* chooses *foo.s*. Like *bsd.prog.mk,* it includes *bsd.man.mk*. Unlike *bsd.prog.mk,* it does this even if NOMAN is defined.

bsd.subdir.mk

bsd.subdir.mk contains definitions for making files in subdirectories. Since only a single program target can be made per directory, BSD-style directory trees tend to have more branches than others, and each program is placed in its own subdirectory. For example, if I have three programs *foo, bar* and *baz*, I might normally write a *Makefile* with the rule

```
all:    foo bar baz

foo:    foo.c foobar.h conf.h

bar:    bar.c foobar.h zot.h conf.h

baz:    baz.c baz.h zot.h conf.h
```

As we have seen, this is not easy to do with the BSD configuration scheme. Instead, you might place all the files necessary to build *foo* in the subdirectory *foo*, and so on. You could then write

```
SUBDIRS = foo bar baz
.include <bsd.subdir.mk>
```

foo/Makefile could then contain

```
PROG  = foo
DPADD = foo.c foobar.h conf.h
.include <bsd.prog.mk>
```

bsd.subdir.mk is structured in the same way as *bsd.prog.mk*. Use *bsd.prog.mk* for making files in the same directory, and *bsd.subdir.mk* for making files in subdirectories. If you want to do both, use *bsd.prog.mk* and define SUBDIR instead of SUBDIRS.

bsd.man.mk

bsd.man.mk contains definitions for installing man pages. It is included from *bsd.prog.mk* and *bsd.lib.mk*, so the target and variables are available from both of these files as well. It defines the target maninstall, which installs the *man* pages and their links, and uses or defines the

A *profiling library* is a library that contains additional code to aid profilers, programs that analyze the CPU usage of the program. We don't cover profiling in this book.

variables described in Table 19-6:

Table 19−6: Variables defined or used by bsd.man.mk

Variable	Meaning
`MANDIR`	The base path of the installed *man* pages. Defaults to */usr/share/man/cat*. The section number is appended directly to `MANDIR`, so that a man page *foo.3* would be installed in */usr/share/man/cat3/foo.3*.
`MANGRP`	The group that owns the *man* pages. Defaults to *bin*.
`MANOWN`	The owner of the *man* pages. Defaults to *bin*.
`MANMODE`	The permissions of the installed *man* pages. Defaults to 444 (read permission for anybody).
`MANSUBDIR`	The subdirectory into which to install machine specific *man* pages. For example, i386 specific pages might be installed under */usr/share/man/cat4/i386*. In this case, `MANSUBDIR` would be set to `/i386`.
`MANn`	(`n` has the values 1 to 8). Manual page names, which should end in `.[1-8]`. If no `MANn` variable is defined, `MAN1=${PROG}.1` is assumed.
`MLINKS`	A list of pairs of names for manual page links. The first filename in a pair must exist, and it is linked to the second name in the pair.

bsd.own.mk

Not all variants of the BSD configuration system use*bsd.own.mk*. Where it is supplied, it contains default permissions, and may be used to override definitions in *sys.mk*, which *include*s it.

bsd.doc.mk

bsd.doc.mk contains definitions for formatting hardcopy documentation files. It varies significantly between versions and omits even obvious things like formatting the document. It does, however, define the variables in Table 19-7, which can be of use in your own *Makefile*:

Table 19−7: Variables defined in bsd.doc.mk

Variable	Meaning
`PRINTER`	Not a printer name at all, but an indicator of the kind of output format to be used. This is the argument to the *troff* flag `-T`. Defaults to `ps` (PostScript output).
`BIB`	The name of the *bib* processor. Defaults to `bib`.
`COMPAT`	Compatibility mode flag for groff when formatting documents with Berkeley me macros. Defaults to `-C`.

Table 19−7: Variables defined in bsd.doc.mk (continued)

Variable	Meaning
`EQN`	How to invoke the *eqn* processor. Defaults to `eqn -T${PRINTER}`.
`GREMLIN`	The name of the gremlin processor. Defaults to `grn`.
`GRIND`	The name of the *vgrind* processor. Defaults to `vgrind -f`.
`INDXBIB`	Name of the *indxbib* processor. Defaults to `indxbib`.
`PAGES`	Specification of the page range to output. Defaults to `1-`.
`PIC`	Name of the *pic* processor. Defaults to `pic`.
`REFER`	Name of the *refer* processor. Defaults to `refer`.