13. make

Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

make(1) provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways.

An individual programmer can easily forget such things as:

- file-to-file dependencies
- files that were modified and the impact that has on other files
- the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to:

- find the target in the description file
- ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile**, **Makefile**, or **s.[mM]akefile**. If this naming convention is followed, the simple command **make** is usually enough to regenerate the target regardless of the number files edited since the last **make**. Usually, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, typing the **make** command rather than typing all the commands to regenerate the target ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all the files on which the target file depends exist and are up to date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file
- filenames and last-modified times from the file system
- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the **math** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line:

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lm  -o  prog

x.o  y.o :   defs.h
```

If this information were stored in a file named **makefile**, the command:

```
make
```

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc –c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o :   x.c  defs.h
        cc  -c  x.c
y.o :   y.c  defs.h
        cc  -c  y.c
z.o :   z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command:

make x.o

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions" below.) Thus, a "save" entry might be included to copy a certain set of files, or a "clean" entry might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by what the macro stands for. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

The **$***, **$@**, **$?**, and **$<** are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under "Description Files and Substitutions.") The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command:

```
make LIBES="-ll -lm"
```

loads the three objects with both the **lex** (**-ll**) and the **math** (**-lm**) libraries, because macro definitions on the command line override definitions in the description file. (In operating system commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:  $(OBJECTS)
        $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        @size make

$(OBJECTS):  defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

              # print files that are out-of-date
              # with respect to "print" file.

print: $(FILES)
       pr $? | $(LP)
       touch print
```

The **make** program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```
cc  -O -c main.c
cc  -O -c doname.c
cc  -O -c misc.c
cc  -O -c files.c
cc  -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -O -c gram.c
cc  main.o doname.o misc.o files.o dosys.o
     gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an at sign, @, in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that a sharp, #, and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp are totally ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (See Figure 13-2 at the end of the chapter.)

General Form

The general form of an entry in a description file is:

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
 . . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp, #, except when the sharp is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon

form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries" section later in this chapter.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (**-s** option of the **make** command) or if the command line in the description file begins with an @ sign. **make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **-i** flag has been specified on the **make** command line, if the fake target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name DEFAULT are used. If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)** (see below).

Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single character macro. The **F** refers to the filename part of the single character macro. These additions are useful when building

hierarchical **makefiles**. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a command can be:

```
cd $(<D); $(MAKE) $(<F)
```

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script, which can handle all the C language programs (i.e., those files ending in **.c**). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Recursive Makefiles

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line, the line is executed even if the **–n** flag is set. Since the **–n** flag is exported across invocations of **make** (through the MAKEFLAGS variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile**(s) describes a set of software subsystems. For testing purposes, **make –n ...** can be executed and everything that would have been done will be printed including output from lower level invocations of **make**.

Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **–r** flag is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **.r** file to a **.o** file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **$*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **$<** is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependents are added to the usual list. A .SUFFIXES line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

- **.o** Object file
- **.c** C source file
- **.c~** SCCS C source file
- **.f** FORTRAN source file
- **.f~** SCCS FORTRAN source file
- **.s** Assembler source file
- **.s~** SCCS Assembler source file
- **.y** **yacc** source grammar
- **.y~** SCCS **yacc** source grammar

.l **lex** source grammar

.l~ SCCS **ex** source grammar

.h Header file

.h~ SCCS header file

.sh Shell file

.sh~ SCCS shell file

Figure 13-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

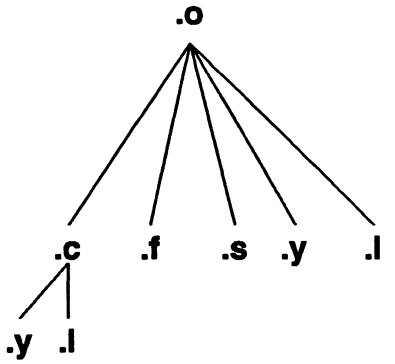

Figure 13-1. Summary of Default Transformation Path

If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 13-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, F77, YACC, and LEX. The command:

```
make CC=newcc
```

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASFLAGS, CFLAGS, F77FLAGS, YFLAGS, and LFLAGS

may be set to cause these commands to be issued with optional flags. Thus, the command line:

make "CFLAGS=-g"

causes the **cc** command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library as follows:

```
projlib(object.o)
```

or

```
projlib((entrypt))
```

where the second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object filename.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::   projlib(pfile1.o)
        $(CC) -c -O pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib::   projlib(pfile2.o)
        $(CC) -c -O pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

... and so on for each object ...

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the filename being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde, ~, syntax will be described shortly.) The user may define other needed rules in the description file.

The above two-member library is then maintained with the following shorter **makefile**:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
                @echo projlib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file; and the filename minus the suffix, respectively (**pfile1.c** and **pfile1**). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction:

```
projlib:    projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to **pfile1.c**. Also, there is no **pfile1.o** file.

1. **make projlib.**
2. Before **make**ing **projlib**, check each dependent of **projlib**.
3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.
4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)
5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.
6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **$@** (=**projlib**) and **$*** (=**pfile1**).
7. Look for a rule **.X.a** and a file **$*.X**. The first **.X** (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set **$<** to be **pfile1.c** and execute the rule. **make** must then compile **pfile1.c**.

8. The library has been updated. Execute the command associated with the **projlib:** dependency, namely:

```
@echo projlib up-to-date
```

Note that to let **pfile1.o** have dependencies, the following syntax is required:

```
projlib(pfile1.o):          $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parenthesis.

Source Code Control System Filenames: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the filename part of the complete path name.

To allow **make** easy access to the prefix **s.** the tilde, ~, is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is:

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

```
.c~
.f~
.y~
.l~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

.c~:
.f~:
.sh~:
.c~.a:
.c~.c:
.c~.o:
.f~.a:
.f~.f:
.f~.o:
.s~.a:
.s~.s:
.s~.o:
.y~.c:
.y~.o:
.l~.l:
.l~.o:
.h~.h:

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS filename format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the operating system program **cat**, a rule in the **makefile** of the following form is needed:

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

In fact, this **.c:** rule is internally defined so no **makefile** is necessary at all. The user only needs to type:

make cat dd echo date

(these are all operating system single-file programs) and all four C language source files are passed through the above shell command line associated with the **.c:** rule.

The internally defined single suffix rules are:

.c:
.c~:
.f:
.f~:
.sh:
.sh~:

Others may be added in the **makefile** by the user.

include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** will read. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **includes** are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named **s.makefile** or **s.Makefile** exists, **make** will do a **get** on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists, which allows access to the file part of **$@**. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file.

For instance, the operating system command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile**.

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like:

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make**(1) in the *Programmer's Reference Manual.*

The make Command

The **make** command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

make [*options*] [*macro definitions*] [*targets*]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.

–t Touch the target files (causing them to be up to date) rather than issue the usual commands.

–q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

–p Print out the complete set of macro definitions and target descriptions.

–k Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.

–e Environment variables override assignments within **makefile**s.

–f Description filename. The next argument is assumed to be the name of a description file. A filename of – denotes the standard input. If there are no **–f** arguments, the file named **makefile** or **Makefile** or **s.[mM]akefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following two arguments are evaluated in the same way as flags:

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.PRECIOUS Dependents on this target are not removed when quit or interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A macro, MAKEFLAGS, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the **makefile** update MAKEFLAGS. Thus, to describe how the environment interacts with **make**, the MAKEFLAGS macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **–f**, **–p**, and **–r** flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).
4. Read the **makefile**(s). The assignments in the **makefile**(s) overrides the environment. This order is chosen so that when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the **–e** flag is used. The **–e** is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make –e ...** is typed, the variables in the environment override the definitions in the **makefile**. Also MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions
2. environment
3. **makefile**(s)
4. command line

The **–e** flag has the effect of rearranging the order to:

1. internal definitions
2. **makefile**(s)
3. environment
4. command line

This order is general enough to allow a programmer to define a **makefile** or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a line that specifies:

```
#include "defs.h"
```

then the object file **x.o** depends on **defs.h**; the source file **x.c** does not. If **defs.h** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the **–n** option is very useful. The command:

make –n

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a comment to an **include** file), the **–t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command:

make –ts

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **make** are reproduced below.

```
#
#       SUFFIXES RECOGNIZED BY MAKE
#
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~
#
#       PREDEFINED MACROS
#
MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
```

Figure 13-2. make Internal Rules (Sheet 1 of 5)

```
#
#       SINGLE SUFFIX RULES
#
.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.c~:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
        -rm -f $*.c
.f:
        $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@
.f~:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
        -rm -f $*.f
.sh:
        cp $< $@; chmod 0777 $@
.sh~:
        $(GET) $(GFLAGS) $<
        cp $*.sh $*; chmod 0777 $@
        -rm -f $*.sh
```

Figure 13-2. make Internal Rules (Sheet 2 of 5)

```
#
#       DOUBLE SUFFIX RULES
#
.c~.c  .f~.f  .s~.s  .sh~.sh  .y~.y  .l~.l  .h~.h:
        $(GET) $(GFLAGS) $<

.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o

.c~.a:
        $(GET) $(GFLAGS) $<
        $(CC) -c $(CFLAGS) $*.c
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[co]

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

.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c

.f.a:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.o

.f~.a:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[fo]
```

Figure 13-2. make Internal Rules (Sheet 3 of 5)

```
.f.o:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
.f~.o:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        -rm -f $*.f
.s~.a:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[so]
.s.o:
        $(AS) $(ASFLAGS) -o $@ $<
.s~.o:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -rm -f $*.s
.l.c :
        $(LEX) $(LFLAGS) $<
        mv lex.yy.c $@
.l~.c:
        $(GET) $(GFLAGS) $<
        $(LEX) $(LFLAGS) $*.l
        mv lex.yy.c $@
```

Figure 13-2. make Internal Rules (Sheet 4 of 5)

```
.l.o:
        $(LEX) $(LFLAGS) $<
        $(CC) $(CFLAGS) -c lex.yy.c
        rm lex.yy.c
        mv lex.yy.o $@
        -rm -f $*.l

.l~.o:
        $(GET) $(GFLAGS) $<
        $(LEX) $(LFLAGS) $*.l
        $(CC) $(CFLAGS) -c lex.yy.c
        rm -f lex.yy.c $*.l
        mv lex.yy.o $*.o

.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@

.y~.c :
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        mv y.tab.c $*.c
        -rm -f $*.y

.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@

.y~.o:
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        $(CC) $(CFLAGS) -c y.tab.c
        rm -f y.tab.c $*.y
        mv y.tab.o $*.o
```

Figure 13-2. make Internal Rules (Sheet 5 of 5)