Making Files

The Make Utility

Many types of files are dependent on various other files in their creation. If the files that make up the final product are updated, the final product becomes out-of-date. The make utility is designed to automate the maintenance and re-creation of files that change over a period of time.

make maintains the files by using a special type of procedure file known as a ***makefile***. The makefile describes the relationship between the final product and the files that make up the final product. For the purpose of this discussion, the final product is referred to as the ***target file*** and the files that make up the target file are referred to as ***dependents***.

+ A makefile contains three types of entries:

- Dependency entries
- Command entries
- Comment entries

¿ A dependency entry specifies the relationship of a target file and the dependents used to build the target file. The entry has the following syntax:

```
<target>:[[<dependent>],<dependent>]
```

The list of files following the target file is known as the ***dependency list***. Any number of dependents can be listed in the dependency list. Any number of dependency entries can be listed in a makefile. A dependent in one entry may also be a target file in another entry. There is, however, only one main target file in each makefile. The main target file is usually specified in the first dependency entry in the makefile.

¡ A command entry specifies the particular command that must be executed to update, if necessary, a particular target file. make updates a target file only if its dependents are newer than itself. If no instructions for update are provided, make attempts to create a command entry to perform the operation.

make recognizes a command entry by a line beginning with one or more spaces or tabs. Any legal OS-9 command line is acceptable. More than one command entry can be given for any dependency entry. Each command entry line is assumed to be complete unless it is continued from the previous command with a backslash (\). Comments should not be interspersed with commands. For example:

```
<target>:[[<file>],<file>]
    <OS-9 command line>
    <OS-9 command line>\
    <continued command line>
```

¬ A comment entry consists of any line beginning with an asterisk (*). All characters following a pound sign (#) are also ignored as comments unless a digit immediately follows the pound sign. In this case, the pound sign is considered part of the command entry. All blank lines are ignored. For example:

```
<target>:[[<file>],<file>]

    * the following command will be executed if the dependent
    * files are newer than the target file
    <OS-9 command line>  # this is also a comment
```

Any entry may be continued on the following line by placing a space followed by a backslash (\) at the end of the line to be continued. All entries longer than 256 characters must be continued on another line. All continuation lines must adhere to the rules for its type of entry. For example, if a command line is continued on a second line, the second line must begin with a space or a tab:

```
FILE: aaa.r bbb.r ccc.r ddd.r eee.r \
fff.r ggg.r
       touch aaa.r bbb.r ccc.r \
       ddd.r eee.r fff.r ggg.r
```

NOTE: Spaces and tabs preceding non-command, continuation lines are ignored.

+ To run the make utility, type make, followed by the name of the file(s) to be created and any options desired.

make processes the makefile three times.

During the first pass, make examines the makefile and sets up a table of dependencies. This table of dependencies stores the target file and the dependency files exactly as they are listed in the makefile. When make encounters a name on the left side of a colon, it first checks to see if it has encountered the name before. If it has, make connects the lists and continues.

After reading the makefile, make determines the target file on the list. It then makes a second pass through the dependency table. During this pass, make tries to resolve any existing ***implicit dependencies***. Implicit dependencies are discussed below.

make does a third pass through the list to get and compare the file dates. When make finds a file in a dependency list that is newer than its target file, it executes the specified command(s). If no command entry is specified, make generates a command based on the assumptions given in the next section. Because OS-9 only stores the time down to the closest minute, make re-makes a file if its date matches one of its dependents.

When a command is executed, it is echoed to standard output. make normally stops if an error code is returned when a command line is executed.

To understand the relationship of the target file, its dependents, and the commands necessary to update the target file, the structure of the makefile must be carefully examined.

Implicit Definitions

Any time a command line is generated, make assumes that the target file is a program to compile. Therefore if the target file is not a program to compile, any necessary command entries must be specified for each dependency list. make uses the following definitions and rules when forced to create a command line.

object files:	Files with no suffixes. An object file is made from a relocatable file and is linked when it needs to be made.
relocatable files:	Files appended by the suffix: .r. Relocatable files are made from source files and are assembled or compiled if they need to be made.
source files:	Files having one of the following suffixes: .a, .c, .f, or .p.
default compiler:	cc
default assembler:	r68

default linker:	cc
default directory for all files:	current data directory (.)

NOTE: The default linker should only be used with programs using Cstart.

Macro Recognition

In addition to recognizing compilation rules and definitions, make recognizes certain macros. make recognizes a macro by the dollar sign ($) character in front of the name. If a macro name is longer than a single character, the entire name must be surrounded by parentheses. For example, $R refers to the macro R, $(PFLAGS) refers to the macro PFLAGS, $(B) and $B refer to the macro B, and $BR is interpreted as the value for the macro B followed by the character R.

Macros may be placed in the makefile for convenience or on the command line for flexibility. Macros are allowed in the form of <macro name> = <expansion>. The expansion is substituted for the macro name whenever the macro name appears.

+ If you define a macro in your makefile and then redefine it on the command line, the command line definition overrides the definition in the makefile. This feature is useful for compiling with special options.

To increase make's flexibility, special macros can be defined in the makefile. make uses these macros when assumptions must be made in generating command lines or when searching for unspecified files. For example, if no source file is specified for program.r, make searches either the directory specified by SDIR or the current data directory for program.a (or .c, .p, .f).

make recognizes the following special macros:

Macro	Definition
ODIR=<path>	make searches the directory specified by <path> for all files with no suffix or relative pathlist. If ODIR is not defined in the makefile, make searches the current directory by default.
SDIR=<path>	make searches the directory specified by <path> for all source files not specified by a full pathlist. If SDIR is not defined in the makefile, make searches the current directory by default.
RDIR=<path>	make searches the directory specified by <path> for all relocatable files not specified by a full pathlist. If RDIR is not defined, make searches the current directory by default.

Macro	Definition
CFLAGS=<opts>	These compiler options are used in any necessary compiler command lines.
RFLAGS=<opts>	These assembler options are used in any necessary assembler command lines.
LFLAGS=<opts>	These linker options are used in any necessary linker command lines.
CC=<comp>	make uses this compiler when generating command lines. The default is cc.
RC=<asm>	make uses this assembler when generating command lines. The default is r68.
LC=<link>	make uses this linker when generating command lines. The default is cc.

Some reserved macros are expanded when a command line associated with a particular file dependency is forked. These macros may only be used on a command line. When you need to be explicit about a command line but have a target program with several dependencies, these macros can be useful. In practice, they are wildcards with the following meanings:

Macro	Definition
$@	Expands to the file name made by the command.
$*	Expands to the prefix of the file to be made.
$?	Expands to the list of files that were found to be newer than the target on a given dependency line.

Make Generated Command Lines

make can generate three types of command lines: compiler command lines, assembler command lines and linker command lines.

¿ Compiler command lines are generated if a source file with a suffix of .c, .p or .f needs to be recompiled. The compiler command line generated by make has the following syntax:

```
$(CC) $(CFLAGS) -r=$(RDIR) $(SDIR)/<file>[.c, .f, or .p]
```

¡ Assembler command lines are generated when an assembly language source file needs to be re-assembled. The assembler command line generated by make has the following syntax:

```
$(RC) $(RFLAGS) $(SDIR)/<file>.a -o=$(RDIR)/<file>.r
```

¬ Linker command lines are generated if an object file needs to be relinked in order to re-make the program module. The linker command line generated by make has the following syntax:

```
$(LC) $(LFLAGS) $(RELS)/<file>.r -f=$(ODIR)/<file>
```

WARNING: When make is generating a command line for the linker, it looks at its list and uses the first relocatable file that it finds, but only the first one. For example:

```
prog: x.r y.r z.r
```

generates

cc x.r, not **cc x.r y.r z.r** or **cc prog.r**

Make Options

Several options allow make even greater versatility for maintaining files/modules. These options may be included on the command line when you run make or they may be included in the makefile for convenience.

When a command is executed, it is echoed to standard output, unless the -s, or silent, option is used or the command line starts with an "at" sign (@). When the -n option is used, the command is echoed to standard output but not actually executed. This is useful when building your original makefile.

make normally stops if an error code is returned when a command line is executed. Errors are ignored if the -i option is used or if a command line begins with a hyphen.

Sometimes, it is helpful to see the file dependencies and the dates associated with each of the files in the list. The -d option turns on the make debugger and gives a complete listing of the macro definitions, a listing of the files as it checks the dependency list and all the file modification dates. If it cannot find a file to examine its date, it assumes a date of -1/00/00 00:00, indicating the necessity to update the file.

If you want to update the date on a file, but do not want to remake it, you can use the -t option. make merely opens the file for update and then closes it, thus making the date current.

If you are quite explicit about your makefile dependencies and do not want make to assume anything, you may use the -b option to turn off the built-in rules governing implicit file dependencies.

Options	Description
-?	Displays the options, function, and command syntax of make.
-b	Does not use built in rules.
-bo	Does not use built in rules for object files.
-d	Prints the dates of the files in makefile (Debug mode).
-dd	Double debug mode. Very verbose.
-f-	Reads the makefile from standard input.
-f=<path>	Specifies <path> as the makefile. If <path> is specified as a hyphen (-), make commands are read from standard input.
-i	Ignores errors.
-n	Does not execute commands, but does display them.
-s	Silent Mode: executes commands without echo.
-t	Updates the dates without executing commands.
-u	Does the make regardless of the dates on files.
-x	Uses the cross-compiler/assembler.
-z	Reads a list of make targets from standard input.
-z=<path>	Reads a list of make targets from <path>.

Examples of the Make Utility

The rest of this chapter is designed to show you different ways to maintain programs with make. These examples are not meant to be totally inclusive of the ways in which make can be used.

Example One: Updating a Document

The following example shows how make can be used to maintain current documentation that is made up of different sections:

```
utils.man: chap1 chap2 apdx
    del utils.man.old;rename utils.man utils.man.old
    merge chap1 chap2 apdx >utils.man
chap1: c1a c1b c1c c1d
    del chap1.old rename chap1 chap1.old
    list c1a c1b c1c c1d ! lxfilter >chap1
chap2: c2a c2b c2c
    del chap2.old rename chap2 chap2.old
    list c1a c1b c1c c1d ! lxfilter >chap1
apdx: functions header footer
    del apdx.old rename apdx apdx.old
    qsort functions >/pipe/func
    list header /pipe/func footer ! lxfilter >apdx
```

The above makefile creates the file utils.man. utils.man is created from three files: chap1, chap2, and apdx. Each of these files is in turn created from the files listed in their dependency lists.

If chap1, chap2, and/or apdx have dependencies with a more recent date, the commands following their respective dependency entries are executed. If chap1, chap2, and/or apdx are re-created, the commands following the initial dependency entry are executed.

Example Two: Compiling C Programs

In this example, make is used to compile high level language modules. Each command and dependency is specified.

```
program: xxx.r yyy.r
   cc xxx.r yyy.r -xf=program
xxx.r: xxx.c /d0/defs/oskdefs.h
   cc xxx.c -r
yyy.r: yyy.c /d0/defs/oskdefs.h
   cc yyy.c -r
```

This makefile specifies that program is made up of two .r files: xxx.r and yyy.r. These files are dependent upon xxx.c and yyy.c respectively and both are dependent on the oskdefs.h file.

If either xxx.c or /d0/defs/oskdefs.h has a date more recent than xxx.r, the command cc xxx.c -r is executed. If yyy.c or /d0/defs/oskdefs.h is newer than yyy.r, then cc yyy.c -r is executed. If either of the former commands are executed, the command cc xxx.r yyy.r xf=program is also executed.

In this example, make specifies each command it must execute. Often this is unnecessary as make uses specific definitions, macros, and built-in assumptions to facilitate program compilation to generate its own commands.

Refining the C Compiler Example

Knowing how make works and understanding the implicit rules can simplify coding immensely:

```
program: xxx.r yyy.r
  cc xxx.r yyy.r -xf=program
xxx.r yyy.r: /d0/defs/oskdefs
```

The above makefile now exploits make's awareness of file dependencies. No mention is made of the C language files; therefore, make looks in the directory specified by the macro definition SDIR = <path> and adjusts the dependency list accordingly. In this case, make looks in the current directory by default. make also generates a command line to compile xxx.r and yyy.r if one or both needs to be updated.

Further simplification would be possible, if program was made up of only one source file:

```
program:
```

make assumes the following from this simple command:

- program has no suffix. It is an object file and therefore needs to rely on relocatable files to be made.
- No dependency list is given; therefore, make creates an entry in the table for program.r.

- After creating an entry for program.r, make creates the entry for a source file connected to the relocatable file.

Assuming it found program.a, it checks the dates on the various files and generates one or both of the following commands if required:

```
r68 program.a -o=program.r

cc program.r -f=program
```

Example Three: A Makefile that Uses Macros

Using these inherent features of make can be especially helpful if you have several object files you want make to check:

```
* beginning
ODIR = /d0/cmds
RDIR = rels
UTILS = attr copy load dir backup dsave
SDIR = ../utils/sources

utils.files: $(UTILS)
    touch utils.files

* end
```

make looks in rels for attr.r, copy.r, etc. and looks in ../utils/sources for attr.c, copy.c, etc. make then generates the proper commands to compile and/or link any of the programs that need to be made. If one of the files in UTILS is made, the command touch utils.files is forked to maintain a current overall date.

Example Four: Putting It All Together

The following example is a makefile to create make:

```
* beginning
ODIR = /h0/cmds
RDIR = rels
CFILES = domake.c doname.c dodate.c domac.c
RFILES = domake.r doname.r dodate.r
PFLAGS = -p64 -nh1
R2 = ../test/domac.r
RFLAGS = -q
make: $(RFILES) $(R2) getfd.r
   linker
$(RFILES): defs.h
$(R2): defs.h
   cc $*.c -r=../test
print.file: $(CFILES)
   pr $? $(PFLAGS) >-/p1
   touch print.file
*end
```

The makefile in this example looks for the .r files listed in RFILES in the directory specified by RDIR: rels. The only exception is ../test/domac.r, which has a complete pathlist specified.

Even though getfd.r does not have any explicit dependents, its dependency on getfd.a is still checked. The source files are all found in the current directory.

Notice that this makefile can also be used to make listings. By typing make print.file on the command line, make expands the macro $? to include all of the files updated since the last time print.file was updated. If you keep a dummy file called print.file in your directory, make will only print out the newly made files. If no print.file exists, all files are printed.

End of Chapter 6