make Command

February 11, 1998

1 Purpose

Maintains up-to-date versions of programs.

2 Syntax

make [-d] [-e] [-i] [-k] [-n] [-p] [-q] [-r] [-S] [-s] [-t] [-f MakeFile ...] [TargetFile ...]

3 Description

The make command reads MakeFile for information about the specified TargetFiles and for the commands necessary to update them. The make command does not change the TargetFile if you have not changed any of the source files since you last built it. It considers a missing file to be a changed file (out-of-date). You can also include macro definitions on the command line after all of the flags. Macro definitions have the form:

macro-name = string

The make command considers all entries on the command line that follow the flags and that do not contain an equal sign to be target file names. The default shell is determined by:

- The MAKESHELL environment variable, if defined. (BSD users should set this to /usr/bin/sh)
- If MAKESHELL is undefined, the program uses the value of SHELL that you set from the description file or the command line. Note that a SHELL set from the environment is always ignored.
- If MAKESHELL and SHELL are undefined, the program uses the default value /usr/bin/sh.

The make command stores its default rules in the external file /usr/ccs/lib/make.cfg. The /usr/ccs/lib/make.cfg file is a configuration file that contains default rules for the make command. These defaults may be overidden in the user Makefile. You may substitute your own rules file by setting the MAKERULES variable to your own file name from the command line. The following lines show how to change the rules file from the command line:

make MAKERULES=/pathname/filename

4 Description File

4.1 Rules

The make command searches the current directory for the first description file it finds that is named makefile, Makefile, s.makefile, or s.Makefile, in that order. The -f option may be used to override this naming convention. The description file contains a sequence of entries specifying the files that the target files depend on. The general form of an entry is:

target [target] . . . :[:] [File] . . . [‘shell command expression‘]. . . [;Command]
(tab)Command

The first line of an entry (called the dependency line), contains a list of targets followed by a : (colon) and an optional list of prerequisite files or dependencies or shell command expressions, which are commands enclosed in single backquotes (‘) whose output is treated as dependencies. If you put shell commands on the dependency line, they must be preceded by a ; (semicolon). All commands that follow the semicolon and all following lines that begin with a tab contain shell commands that the make command uses to build the target.

To specify more than one set of commands for a TargetFile, you must enter more than one dependency definition. In this case, each definition must have the TargetFile name followed by two colons (::), a dependency list, and a command list.

A # (pound sign) is used to put comments in a description file. All characters following a # and continuing to the end of the line are considered to be part of the comment. Comments may be specified on individual lines as well as on dependency lines and command lines in the description file. The description file can contain blank lines, which will be ignored by the make program. However, no blank lines are allowed in a continued list.

The first line that does not begin with a tab or # (pound sign) begins a new dependency or a macro definition. Command lines are performed one at time, each by its own subshell. Thus, the effect of some shell commands, such as cd, does not extend across new-line characters. You can, however, put a (backslash) at the end of a line to continue it on the next physical line.

Commands are executed by passing the command line to the command interpreter. If commands are executed in the background, the make command does not wait for the termination of the background processes before it exits.

The first one through three characters in a command can be one of the following special characters:

- - Ignores errors returned by the command on this line.
- @ Does not display this command line.
- + Executes this command line even if -n, -q or -t is specified.

4.2 Suffix Rules

The make command has default rules that govern the building of most standard files. These rules depend on the standard suffixes used by the system utility programs to identify file types. These rules define the starting and ending file types so that, for example, given a specified .o file, the make command can infer the existence of a corresponding .c file and knows to compile it using the cc -c command.

A rule with only one suffix (that is, .c:) defines the building of prog from all its source files. Use a ˜ (tilde) in the suffix to indicate an SCCS file. For example, the .c .o rule governs changing an SCCS C source file into an object file. You can define rules within the description file. The make command recognizes as a rule any target that contains no slashes and starts with a dot.

You can also add suffixes to the list of suffixes recognized by the make command and add to the default dependency rules. Use the target name .SUFFIXES followed by the suffixes you want to add. Be careful of the order in which you list the suffixes. The make command uses the first possible name for which both a file and a rule exist. The default list is:

.SUFFIXES: .o .c .c .f .f .y .y .l .l .s .s˜

You can clear the list of suffixes by including .SUFFIXES: with no following list.

4.3 Special Target Names

You can use some special target names in the description file to tell the make command to process the file in a different manner. These are:

.DEFAULT The commands that appear after this name in the description file tell the make command what to do if it can find no commands or default rules to tell it how to create a specific file.

.IGNORE If this name appears on a line by itself, the make command does not stop when errors occur. If this has prerequisites, the make command ignores errors associated with them. Using a - (minus) as the first character on a command line in the description file tells the make command to ignore errors for the command on that line only.

.POSIX The make command processes the makefile as specified by the POSIX standard.

.PRECIOUS The files named on the same line as this special name are not removed when the make command is interrupted. If no file names are specified, all files are treated as if specified with **.PRECIOUS**.

.SILENT If this name appears on a line by itself, the make command does not display any of the commands that it performs to build a file. If this has prerequisites, the make command does not display any of the commands associated with them.

.SUFFIXES Use this name to add more suffixes to the list of file suffixes that make recognizes.

4.4 Environment

When you run the make command, it reads the environment and treats all variables as macro definitions. The make command processes macro definitions in the following order:

1. The make command's own default rules
2. Environment variables
3. Description files
4. Command line.

Therefore, macro assignments in a description file normally override duplicate environment variables. The -e flag instructs the make command to use the environment variables instead of the description file macro assignments.

The make command recognizes the MAKEFLAGS and MFLAGS environment macros, which the user can set only as environment variable macros that can be assigned any make flag value except -f, -p, and -d. The make command uses the value of MFLAGS only if the MAKEFLAGS macro is not defined. When the make program starts, it invokes itself with the options specified with these environment variables, if defined, in addition to those specified on the command line. It passes these options to any command it invokes, including additional invocations of the make command itself. Thus you can perform a make -n recursively on a software system to see what would have been performed. The -n is passed to further copies of the shell that runs the next level of make commands. In this way, you can check all of the description files for a software project without actually compiling the project.

4.5 Macros

Entries of the form String1 = String2 are macro definitions. String2 consists of all characters that occur on a line before a comment character (#) or before a new-line character that is not a continuation line. After this macro definition, the make command replaces each $(String1) in the file with String2.

The make command also handles nested macro names, in the form of $($(String1)). Where the innermost macro name is expanded, then its value is taken as another macro name that must also be expanded. Ten levels of nesting for macro expansion is supported.

You do not have to use the parentheses around the macro name if the macro name is only one character long and there is no substitute sequence (see the next paragraph). If you use the following form, you can also replace characters in the macro string with other characters for one time that you use the macro:

```
$(String1[:Substitute1=[Substitute2]])
```

The optional :Substitute1 = Substitute2 specifies a substitute sequence. If you specify a substitute sequence, the make program replaces each Substitute1 pattern in the named

macro with Substitute2 (if Substitute1 is a suffix in String1). Strings in a substitute sequence begin and end with any of the following: a blank, tab, new-line character, or beginning of line.

Note: Because the make command uses the dollar sign symbol ($) to designate a macro, do not use that symbol in filenames of targets and parents, or in commands in the description file unless you are using a defined make macro.

4.5.1 Internal Macros

The make command has five internal macros. It assigns values to these macros under one or more of the following conditions:

- When it uses an internal rule to build a file
- When it uses a .DEFAULT rule to build a file
- When it uses rules in the description file to build a file

- When the file is a library member.

They are defined as follows:

$* The file name (without the suffix) of the source file.

$@ The full target name of the current target.

$< The source files of an out-of-date module. The make command evaluates this macro when applying inference rules or the .DEFAULT rule. For example:

```
.c.o:
cc -c $<
```

Here, $< refers to the .c file of any out-of-date .o file.

$? The list of out-of-date files. The make command evaluates this macro when it evaluates explicit rules from MakeFile.

$% The name of an archive library member. The make command evaluates this macro only if the target is an archive library member of the form lib(file.o). In this case, $@ evaluates to lib and $% evaluates to the library member, file.o.

You can add an uppercase D or F to indicate "directory part" or "file part", respectively, to all internal macros except for $?. Thus, $(@D) refers to the directory part of the name $@. If there is no directory part, the make command uses ./.

The $$ symbol is the dollar sign in a description file. A single $ is interpreted by the make command as a macro to be evaluated, as in the above definitions.

4.6 Libraries

If a target name contains parentheses, the make command considers it an archive library. The string within parentheses refers to a library member. Thus, lib(File.o) and $(LIB)(File.o) both see an archive library which contains File.o. (You must have already defined the LIB macro.) The expression $(LIB) + (File1.o File2.o) is not legal.

Rules that apply to archive libraries have the form .x.a, where .x is the suffix of the file you want to add to an archive library. For example, .c.a indicates a rule that changes any C source file to a library file member. For the following description file entry:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
@echo lib is now up to date.
```

The default rule used by the make command is:

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

Note: The .x suffix must be different from the suffix of the archive member.

Another, but more limited, example of an archive library maintenance rule follows:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
$(CC) -c $(CFLAGS) $(?:.o=.c)
ar rv lib $?
rm $? @echo lib is now up to date
```

```
.c.a:;
```

This example rule uses a substitute sequence (.o=.c) to replace with .c files all .o files generated by the $? macro.The $? list is the set of object file names (inside lib)with C source files that are out of date. The macro substitution translates .o to .c.

If this rule appears in your description file, it disables the default .c.a: rule, which creates each object file one by one. This type of organization speeds up archive library maintenance, but becomes hard to use if the archive library contains a variety of source programs, for example, assembly and C programs.

5 Flags

-d Displays detailed information about the files and times that make examines (debug mode).

-e Uses environment variables in place of any assignments made within description files. These assignments normally replace environment variables.

-f MakeFile Reads MakeFile for a description of how to build the target file. If you give only a - (minus) for MakeFile, the make command reads standard input. If you do not use the -f flag, the make command searches the current directory for the first description file named makefile, Makefile, s.makefile, or s. Makefile, in that order. You can specify more than one description file by entering the -f flag more than once (with its associated MakeFile parameter).

-i Ignores error codes returned by commands. The make command normally stops if a command returns a nonzero code. Use this flag to compile several modules only if you want the make command to continue when an error occurs in one of the modules.

-k Stops processing the current target if an error occurs, but continues with other branches that do not depend on that target.

-n Displays commands, but does not run them. However, lines beginning with a + (plus sign) are executed. Displays lines beginning with an @ (at sign). If the special target .POSIX is not specified and the command in the description file contains the string $(MAKE), perform another call to the make command. Use this flag to preview the performance of the make command.

-p Displays the complete set of macro definitions and target descriptions before performing any commands.

-q Returns a zero status code if the target file is up-to-date; returns a one status code if the target file is not up-to-date. However, a command lines with the + (plus) sign prefix will be executed.

-r Does not use the default rules. Uses only rules specified in the description file.

-S Terminates the make command if an error occurs. This is the default and the opposite of -k flag. If the -k and -S flags are both specified, the last one evaluated works.

-s Does not display commands or touch messages on the screen as they are performed.

-t Changes only the date of the files, rather than performing the listed commands. Use this flag if you have made only minor changes to a source file that do not affect anything outside of that file.This flag changes the date of all target files that appear on the command line or in the description file. However, command lines beginning with a + (plus sign) are executed.

6 Examples

1. To make the file specified by the first entry in the description file:

```
make
```

2. To display, but not run, the commands that the make command would use to make a file:

```
make -n search.o
```

You may want to do this to verify that a new description file is correct before using it.