## NAME

make − maintain program groups

## SYNOPSIS

**make** [ **−f** makefile ] [ **−p** ] [ **−i** ] [ **−k** ] [ **−s** ] [ **−r** ] [ **−n** ] [ **−b** ] [ **−e** ] [ **−m** ] [ **−t** ] [ **−q** ] [ **−d** ] [ name ] ...

## DESCRIPTION

*Make* executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **−f** option is present, **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** are tried in order. If *makefile* is **−**, the standard input is taken. More than one **−f** makefile argument pair may appear.

*Make* updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out of date.

*Makefile* contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a colon, then a (possibly null) list of prerequisite files or dependencies. Text following a semicolon, and all following lines that begin with a tab, are Shell commands to be executed to update the target. The first line that does not begin with a tab or sharp begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><newline> sequence. Sharp and new-line surround comments.

The following makefile says that 'pgm' depends on two files 'a.o' and 'b.o', and that they in turn depend on '.c' files and a common file 'incl.h'.

```
pgm: a.o b.o
        cc a.o b.o −o pgm
a.o: incl.h a.c
        cc −c a.c
b.o: incl.h b.c
        cc −c b.c
```

Command lines are executed one at a time, each by its own Shell. A line is printed when it is executed unless the **−s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the first character of the command is **@**. The **−n** option specifies printing without execution; however, if the command line has the string "$(MAKE)" in it the line is always executed (see discussion of **MAKEFLAGS** macro below under "Environment"). The **−t** (touch) option updates the modified date of a file without executing any commands.

Commands returning nonzero status normally terminate *make*. If the **−i** option is present, or the entry **.IGNORE:** appears in *makefile*, or if the line specifying the command begins with <tab><hyphen>, the error is ignored. If the **−k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **−b** option allows old makefiles (those written for the old version of make) to run without errors. The difference between the old version of make and this version is that the new version requires all dependency lines to have a (possibly null) command associated with them. The previous version of make assumed if no command was specified explicitly that the command was null. This was contrary to the documentation.

Interrupt and quit cause the target to be deleted unless the target depends on the special name **.PRECIOUS**.

### The Environment

The environment is read by make. All variables are assumed to be macro definitions and

processed as such. The environment variables are processed before any makefile and after the internal rules. Thus, macro assignments in a makefile override environment variables. The −e option causes the environment to override the macro assignments in a makefile.

The **MAKEFLAGS** environment variable is processed by make as containing any legal input option (except **-f**, **-p**, and **-d**) defined for the command line. Further, upon invocation, make "invents" the variable, if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This proves very useful for "supermakes". In fact, as noted above, when the **-n** option is used, the command "$(MAKE)" is executed anyway; hence, one can perform a *make -n* recursively on a whole software system to see what would have been executed. This is because the **-n** is put in **MAKEFLAGS** and passed to further invocations of "$(MAKE)". This is one way of debugging all of the makefiles for a software project without actually doing anything.

**Macros**

Entries of the form

string1 = string2

are macro definitions. Subsequent appearances of *$(string1[:subst1=[subst2]])* are replaced by *string2*. (The parentheses are optional if a single character macro name is used and there is no substitute sequence.) The optional ":subst1=subst2" is an substitute sequence. If it is specified, all non-overlapping occurrences of "subst1" in the named macro are replaced by "subst2". "Strings" (for the purposes of this type of substitution) are delimited by blanks, tabs, newline character and the beginning of a line. An example of the use of the substitute sequence is given under the Libraries heading.

**Internal Macros**

There are five internally maintained macros which are useful for writing rules for building targets.

**$*** The macro **$*** stands for the file name part with the suffix deleted, of the current dependent. It is evaluated only for inference rules.

**$@** The **$@** macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

**$<** The **$<** macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the "thing" which is out of date with respect to the target (i.e. the "manufactured" dependent file name). Thus, in the ".c.o" rule the **$<** macro would evaluate to the ".c" file.

**$?** The **$?** macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target. (Essentially, those "things" which must be rebuilt.)

**$%** The **$%** macro is only evaluated when 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".

An example: a rule for making optimized '.o' files from '.c' files is:

```
.c.o:
        cc −c −O $*.c
```

or

```
.c.o:
        cc −c −O $<
```

Four of the five macros can have alternative forms. When an upper case 'D' or 'F' is appended to any of the four macros the meaning is changed to "directory part" for 'D' and "file part" for

'F'. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, "./" is generated. The only macro excluded from this alternative form is **$?**. The reasons for this are debatable.

**Suffixes**

Certain names (for instance, those ending with '.o') have inferable prerequisites such as '.c', '.s', etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *Make* has *inference* rules which allow building files from other files by examining the suffixes and determining an appropriate *inference rule* to use. The current default inference rules are:

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

(The internal rules for make are contained in the source file "rules.c" for the make program. These rules are expected to be locally modified. To print out the rules compiled into the make on any machine in a form suitable for recompilation the following command is used:

make -fp - 2>/dev/null </dev/null

The only peculiarity in this output is the "(null)" string which printf(3) prints when handed a null string.)

The tilda in the above rules refers to an SCCS file. Thus, the rule ".c~.o" would transform an SCCS C source file into an object file (".o"). Since the "s." of the SCCS files is a prefix it is incompatible with the "suffix" point of view of make. Hence, the tilda is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e. ".c:") is the definition of how to build "x" from "x.c". In effect, the other suffix is null. This is useful for building targets from only one source file. (e.g. shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

**.SUFFIXES**: .o .c .y .l .s

(Here, again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine.) Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

**Inference Rules**

The first example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o −o pgm
a.o b.o: incl.h
```

This is because make has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*, *lex* and *yacc*(1) respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix *.o* from a file with suffix *.c* is specified as an entry with ".c.o:" as the 'target' and no dependents. Shell commands associated with the target define the rule for making a ".o" file from a ".c" file. Any target which has no slashes in it and starts with a dot is identified as a rule and not a true target.

**Libraries**

If a target or dependency name contains parenthesis, it is assumed to be an archive library, the string within parenthesis referring to a member within the library. Thus "lib(file.o)" and "$(LIB)(file.o)" both refer to an archive library which contains "file.o". (This assumes the "LIB" macro has been previously defined.) The expression "$(LIB)(file1.o file2.o)" is not legal. Rules pertaining to archive libraries have the form ".XX.a" where the "XX" is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the "XX" to be different from the suffix of the archive member. Thus, one cannot have "lib(file.o)" depend upon "file.o" explicitly. The most common use of the archive interface is as follows (Here, we assume the source files are all C type source):

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

In fact, the ".c.a" rule listed above is built into make and, thus, is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction 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:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object file names (inside "lib") whose C source files are out of date. The substitution mode translates the ".o" to ".c". (Unfortunately, one cannot as yet transform to ".c~"; however this will come also.) Note also, the disabling of the ".c.a:" rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

**Options and Other Stuff**

The following is a brief description of all options and some special names:

**-f file** Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. The contents of the description files override the built-in rules if they are present.

**-p** Print out the complete set of macro definitions and target descriptions

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

**-k** abandon work on the current entry, but continue on other branches that do not depend on that entry.

**-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. |
| **-b** | Compatability mode for old makefiles. |
| **-e** | Environment variables override assignments within makefiles. |
| **-m** | Print a memory map showing text, data, and stack. This option is a no-op on systems without the *getu(2)* system call. |
| **-t** | Touch the target files (causing them to be up to date) rather than issue the usual commands. |
| **-d** | Debug mode. Print out detailed information on files and times examined. |
| **-q** | Question. The **make** command returns a zero or non-zero status code depending on whether the target file is or is not up to date. |
| **.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 of this target will not be removed when quit or interrupt are hit. |
| **.SILENT:** | Same effect as the -s option. |
| **.IGNORE** | Same effect as the -i option. |

## FILES

makefile, Makefile, s.makefile, s.Makefile

## SEE ALSO

sh(1)

S. I. Feldman *Make − A Program for Maintaining Computer Programs,* Bell Laboratories Computing Science Tech. Rep. No. 57, April, 1977

E. G. Bradford − An Augmented Version of Make, TM 79-5255-1, July, 1979

## BUGS

Some commands return nonzero status inappropriately. Use −i to overcome the difficulty.
Commands that are directly executed by the Shell, notably *cd*(1), are ineffectual across newlines in *make*.
The syntax "(lib(file1.o file2.o file3.o)" is illegal.
lib(file.o) cannot be built from file.o.
The macro $(a:.o=.c~) doesn't work.