NAME

sysV-make – maintain, update, and regenerate groups of programs

SYNOPSIS

/usr/ccs/lib/svr4.make [–f *makefile***] [–eiknpqrst] [** *names* **]**

DESCRIPTION

This is the "vanilla" System V version of **make**. If the environment variable **USE_SVR4_MAKE** is set, then the command **make** will invoke this version of **make**. (See also the **ENVIRONMENT** section.)

make allows the programmer to maintain, update, and regenerate groups of computer programs. **make** executes commands in *makefile* to update one or more target *names* (*names* are typically programs). If the –**f** option is not present, then **makefile**, **Makefile**, and the Source Code Control System (SCCS) files **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 its dependents 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 Missing files are deemed to be outdated.

The following list of four directives can be included in *makefile* to extend the options provided by **make**. They are used in *makefile* as if they were targets:

.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.

.IGNORE: Same effect as the **–i** option.

.PRECIOUS: Dependents of the **.PRECIOUS** entry will not be removed when quit or interrupt are hit.

.SILENT: Same effect as the –**s** option.

The options for **make** are listed below:

–e Environment variables override assignments within makefiles.

–**f** *makefile* Description filename (*makefile* is assumed to be the name of a description file).

–**i** Ignore error codes returned by invoked commands.

–**k** Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

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

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

–**q** Question. **make** returns a zero or non-zero status code depending on whether or not the target file has been updated.

–**r** Do not use the built-in rules.

–**s** Silent mode. Do not print command lines before executing.

−t Touch the target files (causing them to be updated) rather than issue the usual commands.

Creating the makefile

The makefile invoked with the −**f** option is a carefully structured file of explicit instructions for updating and regenerating programs, and 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 '**:**', then a (possibly null) list of prerequisite files or dependencies. Text following a '**;**' and all following lines that begin with a tab are shell commands to be executed to update the target. The first non-empty line that does not begin with a tab or '**#**' begins a new dependency or macro definition. Shell commands may be continued across lines with a backslash-new-line (\-NEWLINE) sequence. Everything printed by make (except the initial TAB) is passed directly to the shell as is. Thus,

```
echo a\
b
```

will produce

```
ab
```

exactly the same as the shell would.

Number-sign (**#**) and NEWLINE surround comments including contained '\−NEWLINE' sequences.

The following makefile says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) 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. The **SHELL** environment variable can be used to specify which shell **make** should use to execute commands. The default is **/usr/bin/sh**. The first one or two characters in a command can be the following: '@', '−', '@−', or '−@'. If '@' is present, printing of the command is suppressed. If '−' is present, **make** ignores an error. A line is printed when it is executed unless the −**s** option is present, or the entry **.SILENT:** is included in *makefile*, or unless the initial character sequence contains a @. The −**n** option specifies printing without execution; however, if the command line has the string **$(MAKE)** in it, the line is always executed (see the discussion of the **MAKEFLAGS** macro in the **make Environment** sub-section below). The −**t** (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate **make**. If the −**i** option is present, if the entry **.IGNORE:** is included in *makefile*, or if the initial character sequence of the command contains '−', 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.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the directive **.PRECIOUS**.

make Environment

The environment is read by **make**. All variables are assumed to be macro definitions and are 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. Suffixes and their associated rules in the makefile will override any identical suffixes in the built-in rules.

The **MAKEFLAGS** environment variable is processed by **make** as containing any legal input option (except –**f** and –**p**) 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 feature proves very useful for "super-makes". In fact, as noted above, when the –**n** option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make** –**n** recursively on a whole software system to see what would have been executed. This result is possible because the –**n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This usage is one way of debugging all of the makefiles for a software project without actually doing anything.

Include Files

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 and will be read by the current invocation, after substituting for any macros.

Macros

Entries of the form *string1* **=** *string2* are macro definitions. *string2* is defined as all characters up to a comment character or an unescaped NEWLINE. 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 a 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 characters, and beginnings of lines. An example of the use of the substitute sequence is shown in the **Libraries** sub-section below.

Internal Macros

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

$*** The macro **$* stands for the filename part of the current dependent with the suffix deleted. 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 module that is outdated with respect to the target (the “manufactured” dependent file name). Thus, in the **.c.o** rule, the **$<** macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc −c −O $*.c
```

or:

```
.c.o:
        cc −c −O $<
```

$?** The **$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are outdated with respect to the target, and essentially those modules that 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**.

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 **$?**.

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 that 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~	.f	.f~	.s	.s~	.sh	.sh~	.C	.C~
.c.a	.c.o	.c~.a	.c~.c	.c~.o	.f.a	.f.o	.f~.a	.f~.f	.f~.o
.h~.h	.l.c	.l.o	.l~.c	.l~.l	.l~.o	.s.a	.s.o	.s~.a	.s~.o
.s~.s	.sh~.sh	.y.c	.y.o	.y~.c	.y~.o	.y~.y	.C.a	.C.o	.C~.a
.C~.C	.C~.o	.L.C	.L.o	.L~.C	.L~.L	.L~.o	.Y.C	.Y.o	.Y~.C
.Y~.o	.Y~.Y								

The internal rules for **make** are contained in the source file **make.rules** for the **make** program. These rules can 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 −pf − 2>/dev/null </dev/null
```

A tilde in the above rules refers to an SCCS file (see **sccsfile**(4)). Thus, the rule **.c~.o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix, it is incompatible with the **make** suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (for example, **.c:**) is the definition of how to build *x* from *x*.**c**. In effect, the other suffix is null. This feature is useful for building targets from only one source file, for example, shell procedures and 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 .c˜ .y .y˜ .l .l˜ .s .s˜ .sh .sh˜ .h .h˜ .f .f˜ .C .C˜ .Y .Y˜ .L .L˜

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 abbreviation is possible 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**(1B). 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 that 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 parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus, **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library that contains **file.o**. (This example assumes the **LIB** macro has been previously defined.) The expression **$(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 by-product 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 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) $(ARFLAGS) $@ $*.o
        rm −f $*.o
```

In fact, the **.c.a** rule listed above is built into **make** and 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) $(ARFLAGS) 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 filenames (inside **lib**) whose C source files are outdated. The substitution mode translates the **.o** to **.c**. (Unfortunately, one cannot as yet transform to **.c~**; however, this transformation may become possible in the future.) Also note 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.

ENVIRONMENT

USE_SVR4_MAKE

If this environment variable is set, then the **make** command will invoke this System V version of **make**. If this variable is not set, then the default version of **make**(1S) is invoked.

USE_SVR4_MAKE can be set as follows (Bourne shell):

$ USE_SVR4_MAKE=“”; export USE_SVR4_MAKE

or (C shell):

% setenv USE_SVR4_MAKE

FILES

[Mm]akefile and **s.[Mm]akefile**	default makefiles
/usr/bin/sh	default shell for **make**
/usr/share/lib/make/make.rules	default rules for **make**

SEE ALSO

cc(1B), **cd**(1), **make**(1S), **sh**(1), **printf**(3S), **sccsfile**(4)

Programming Utilities Guide

NOTES

Some commands return non-zero status inappropriately; use −**i** or the ‘**-**’ command line prefix to overcome the difficulty.

Filenames containing the characters ‘**=**’, ‘**:**’, and ‘**@**’ will not work. 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. You cannot build **lib(file.o)** from **file.o**.