-- --

IEEE P1003.2 Draft 11.2 – September 1991

6.2 `make` — Maintain, update, and regenerate groups of programs

6.2.1 Synopsis

make [−einpqrst][−f *makefile*]... [−k | −S][*macro=name*]...
[*target_name* ...]

6.2.2 Description

The `make` utility can be used as a part of software development to update files 1
that are derived from other files. A typical case is one where object files are 1
derived from the corresponding source files. The `make` utility examines time rela- 1
tionships and updates those derived files (called targets) that have modified times 1
earlier than the modified times of the files (called prerequisites) from which they 1
are derived. A description file (''makefile'') contains a description of the relation- 1
ships between files, and the commands that must be executed to update the tar- 1
gets to reflect changes in their prerequisites. Each specification, or rule, shall 1
consist of a target, optional prerequisites, and optional commands to be executed
when a prerequisite is newer than the target. There are two types of rules:

— Inference rules, which have one target name with at least one period (`.`) and no slash (`/`)

— Target rules, which can have more than one target name

In addition, `make` shall have a collection of built-in macros and inference rules that infer prerequisite relationships to simplify maintenance of programs.

To receive exactly the behavior described in this clause, a portable makefile shall:

— Include the special target `.POSIX` (see 6.2.7.3)

— Omit any special target reserved for implementations (a leading period followed by uppercase letters) that has not been specified by this clause.

The behavior of `make` is unspecified if either or both of these conditions are not 1
met. 1

6.2.3 Options

The `make` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−e` Cause environment variables, including those with null values, to override macro assignments within makefiles.

`−f` *makefile* Specify a different makefile. The argument *makefile* is a pathname of a description file, which is also referred to as the *makefile* . A pathname of "−" shall denote the standard input. There can be multiple instances of this option, and they shall be processed in the order specified. The effect of specifying the same option-argument more than once is unspecified. See 6.2.7.1.

`−i` Ignore error codes returned by invoked commands. This mode is the same as if the special target `.IGNORE` were specified without prerequisites. See 6.2.7.2.

`−k` Continue to update other targets that do not depend on the current target if a nonignored error occurs while executing the commands to bring a target up to date.

`−n` Write commands that would be executed on standard output, but do not execute them. However, lines with a plus-sign (+) prefix shall be executed. In this mode, lines with an at-sign (@) character prefix shall be written to standard output.

`−p` Write to standard output the complete set of macro definitions and target descriptions. The output format is unspecified.

`−q` Return a zero exit value if the target file is up-to-date; otherwise, return an exit value of 1. Targets shall not be updated if this option is specified. However, a command line (associated with the targets) with a plus-sign (+) prefix shall be executed.

`−r` Clear the suffix list and do not use the built-in rules.

`−S` Terminate `make` if an error occurs while executing the commands to bring a target up-to-date. This shall be the default and the opposite of `−k`.

`−s` Do not write command lines or touch messages (see `−t`) to standard output before executing. This mode shall be the same as if the special target `.SILENT` were specified without prerequisites. See 6.2.7.2.

`−t` Update the modification time of each target as though a `touch` *target* had been executed. See `touch` in 4.63. Targets that have prerequisites but no commands (see 6.2.7.3), or that are already up-to-date, shall not be touched in this manner. Write messages to standard output for each target file indicating the name of the file and that it was touched. Normally, the command lines associated with each target are not executed. However, a command line with a plus-sign (+) prefix shall be executed.

If the `−k` and `−S` options are both specified on the command line, by the **MAKEFLAGS** environment variable, or by the `MAKEFLAGS` macro, the last one evaluated shall take precedence. The **MAKEFLAGS** environment variable shall be evaluated first and the command line shall be evaluated second. Assignments to the `MAKEFLAGS` macro shall be evaluated as described in 6.2.5.3.

6.2.4 Operands

The following operands shall be supported by the implementation:

target_name Target names, as defined in 6.2.7. If no target is specified, while `make` is processing the makefiles, the first target that `make` encounters that is not a special target or an inference rule shall be used.

macro=name Macro definitions, as defined in 6.2.7.4.

If the *target_name* and *macro=name* operands are intermixed on the command line, the results are unspecified.

6.2.5 External Influences

6.2.5.1 Standard Input

The standard input shall be used only if the *makefile* option-argument is –. See Input Files.

6.2.5.2 Input Files

The input file, otherwise known as the makefile, is a text file containing rules, 1
macro definitions, and comments. (See 6.2.7.) 1

6.2.5.3 Environment Variables

The following environment variables shall affect the execution of `make`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

LC_MESSAGES This variable shall determine the language in which messages should be written.

MAKEFLAGS This variable shall be interpreted as a character string representing a series of option characters to be used as the default options. The implementation shall accept both of the following formats (but need not accept them when intermixed):

(1) The characters are option letters without the leading hyphens or <blank> separation used on a command line.

(2) The characters are formatted in a manner similar to a portion of the `make` command line: options are preceded by hyphens and <blank>-separated as described in 2.10.2. The *macro*=*name* macro definition operands can also be included. The difference between the contents of **MAKEFLAGS** and the command line is that the contents of the variable shall not be subjected to the word expansions (see 3.6) associated with parsing the command line values.

When the command-line options `−f` or `−p` are used, they shall take effect regardless of whether they also appear in **MAKEFLAGS**. If they otherwise appear in **MAKEFLAGS**, the result is undefined.

The **MAKEFLAGS** variable shall be accessed from the environment before the makefile is read. At that time, all of the options (except `−f` and `−p`) and command-line macros not already included in `MAKEFLAGS` shall be added to the `MAKEFLAGS` macro. The `MAKEFLAGS` macro shall be passed into the environment as an environment variable for all child processes. If the `MAKEFLAGS` macro is subsequently set by the makefile, it shall replace the **MAKEFLAGS** variable currently found in the environment.

The value of the **SHELL** environment variable shall not be used as a macro and shall not be modified by defining the `SHELL` macro in a makefile or on the command line. All other environment variables, including those with null values, shall be used as macros, as defined in 6.2.7.4.

6.2.5.4 Asynchronous Events

If not already ignored, `make` shall trap SIGHUP, SIGTERM, SIGINT, and SIGQUIT and remove the current target unless the target is a directory or the target is a prerequisite of the special target `.PRECIOUS` or unless one of the `−n`, `−p`, or `−q` options was specified. Any targets removed in this manner shall be reported in diagnostic messages of unspecified format, written to standard error. After this cleanup process, if any, `make` shall take the standard action for all other signals; see 2.11.5.4.

6.2.6 External Effects

6.2.6.1 Standard Output

The `make` utility shall write all commands to be executed to standard output unless the `-s` option was specified, the command is prefixed with an at-sign, or the special target `.SILENT` has either the current target as a prerequisite or has no prerequisites. If `make` is invoked without any work needing to be done, it shall write a message to standard output indicating that no action was taken.

6.2.6.2 Standard Error

Used only for diagnostic messages.

6.2.6.3 Output Files

None. However, utilities invoked by `make` may create additional files.

6.2.7 Extended Description

The `make` utility attempts to perform the actions required to ensure that the specified target(s) are up-to-date. A target is considered out-of-date if it is older than any of its prerequisites or if it does not exist. The `make` utility shall treat all prerequisites as targets themselves and recursively ensure that they are up-to-date, processing them in the order in which they appear in the rule. The `make` utility shall use the modification times of files to determine if the corresponding targets are out-of-date. (See 2.9.1.6.)

After `make` has ensured that all of the prerequisites of a target are up-to-date, and if the target is out-of-date, the commands associated with the target entry shall be executed. If there are no commands listed for the target, the target shall be treated as up-to-date.

6.2.7.1 Makefile Syntax

A makefile can contain rules, macro definitions (see 6.2.7.4), and comments. There are two kinds of rules: inference rules (6.2.7.5) and target rules (6.2.7.3). The `make` utility shall contain a set of built-in inference rules. If the `-r` option is present, the built-in rules shall not be used and the suffix list shall be cleared. Additional rules of both types can be specified in a makefile. If a rule or macro is defined more than once, the value of the rule or macro shall be that of the last one specified. Comments start with a number-sign (#) and continue until an unescaped `<newline>` is reached.

By default, the file `./makefile` shall be used. If `./makefile` is not found, the file `./Makefile` shall be tried. If neither `./makefile` nor `./Makefile` are found, other implementation-defined pathnames may also be tried.

The −f option shall direct `make` to ignore `./makefile` and `./Makefile` (and any implementation-defined variants) and use the specified argument as a makefile instead. If the − argument is specified, standard input shall be used.

The term *makefile* is used to refer to any rules provided by the user whether in `./makefile`, `./Makefile`, or specified by the −f option.

The rules in makefiles shall consist of the following types of lines: target rules, including special targets (see 6.2.7.3); inference rules (see 6.2.7.5); macro definitions (see 6.2.7.4); empty lines; and comments. Comments start with a number sign (#) and continue until an unescaped <newline> is reached.

When an escaped <newline> (one preceded by a backslash) is found anywhere in the makefile, it shall be replaced, along with any leading white space on the following line, with a single <space>.

6.2.7.2 Makefile Execution

Command lines shall be processed one at a time by writing the command line to the standard output (unless one of the conditions listed below under ''@'' suppresses the writing) and executing the command(s) in the line. A <tab> character may precede the command to standard output. Commands shall be executed by passing the command line to the command interpreter in the same manner as if the string were the argument to the function in 7.1.1 [such as the *system* () function in the C binding].

The environment for the command being executed shall contain all of the variables in the environment of `make`. The macros from the command line to `make` shall be added to `make`'s environment. Other implementation-defined variables may also be added to `make`'s environment. If any command-line macro has been defined elsewhere, the command-line value shall overwrite the existing value. If the **MAKEFLAGS** variable is not set in the environment in which `make` was invoked, in the makefile, or on the command line, it shall be created by `make`, and shall contain all options specified on the command line except for the −f and −p options. It may also contain implementation-defined options.

By default, when `make` receives a nonzero status from the execution of a command, it terminates with an error message to standard error.

Command lines can have one or more of the following prefixes: a hyphen (−), an at-sign (@), or a plus-sign (+). These modify the way in which `make` processes the command. When a command is written to standard output, the prefix shall not be included in the output.

- If the command prefix contains a hyphen, or the −i option is present, or the special target `.IGNORE` has either the current target as a prerequisite or has no prerequisites, any error found while executing the command shall be ignored.

@ If the command prefix contains an at-sign and the command-line −n option is not specified, or the −s option is present, or the special target `.SILENT` has either the current target as a prerequisite or has no prerequisites, the command shall not be written to standard output before it is executed.

+ If the command prefix contains a plus-sign, this indicates a command line that shall be executed even if `−n`, `−q`, or `−t` is specified.

6.2.7.3 Target Rules

Target rules are formatted as follows:

target [*target* ...]: [*prerequisite* ...][;*command*] 1
[<tab>*command* 1
<tab>*command* 1
...] 1

(*line that does not begin with* <tab>) 1

Target entries are specified by a <blank>-separated, nonnull list of targets, then a colon, then a <blank>-separated, possibly empty list of prerequisites. Text fol-
lowing a semicolon, if any, and all following lines that begin with a <tab>, are 1
command lines to be executed to update the target. The first nonempty line that 1
does not begin with a <tab> or # shall begin a new entry. An empty or blank 1
line, or a line beginning with #, may begin a new entry. 1

Applications shall select target names from the set of characters consisting solely of periods, underscores, digits, and alphabetics from the portable character set (see 2.4). Implementations may allow other characters in target names as exten-
sions. The interpretation of targets containing the characters ‘‘%’’ and ‘‘"’’ is 1
implementation defined. 1

A target that has prerequisites, but does not have any commands, can be used to add to the prerequisite list for that target. Only one target rule for any given target can contain commands.

Lines that begin with one of the following are called *special targets* and control the operation of `make`:

`.DEFAULT` If the makefile uses this special target, it shall be specified with commands, but without prerequisites. The commands shall be used by `make` if there are no other rules available to build a target.

`.IGNORE` Prerequisites of this special target are targets themselves; this shall cause errors from commands associated with them to be ignored in the same manner as specified by the `−i` option. Subsequent occurrences of `.IGNORE` shall add to the list of targets ignoring command errors. If no prerequisites are specified, `make` shall behave as if the `−i` option had been specified and errors from all commands associated with all targets shall be ignored.

`.POSIX` This special target shall be specified without prerequisites or commands. If it appears before the first noncomment line in the makefile, `make` shall process the makefile as specified by this clause; otherwise, the behavior of `make` is unspecified.

`.PRECIOUS` Prerequisites of this special target shall not be removed if `make` receives one of the asynchronous events explicitly described in 6.2.5.4. Subsequent occurrences of `.PRECIOUS` shall add to the list of precious files. If no prerequisites are specified, all targets in the makefile shall be treated as if specified with `.PRECIOUS`.

`.SILENT` Prerequisites of this special target are targets themselves; this shall cause commands associated with them to not be written to the standard output before they are executed. Subsequent occurrences of `.SILENT` shall add to the list of targets with silent commands. If no prerequisites are specified, `make` shall behave as if the `−s` option had been specified and no commands or touch messages associated with any target shall be written to standard output.

`.SUFFIXES` Prerequisites of `.SUFFIXES` shall be appended to the list of known suffixes and are used in conjunction with the inference rules (see 6.2.7.5). If `.SUFFIXES` does not have any prerequisites, the list of known suffixes shall be cleared. Makefiles shall not associate commands with `.SUFFIXES`.

Targets with names consisting of a leading period followed by the uppercase letters `POSIX` and then any other characters are reserved for future standardization. Targets with names consisting of a leading period followed by one or more uppercase letters are reserved for implementation extensions.

6.2.7.4 Macros

Macro definitions are in the form:

string1 = [*string2*] 1

The macro named *string1* is defined as having the value of *string2* , where *string2* is defined as all characters, if any, after the equals-sign, up to a comment charac- 1
ter (#) or an unescaped `<newline>` character. Any `<blank>`s immediately before or after the equals-sign shall be ignored.

Subsequent appearances of `$(`*string1* `)` or `${`*string1* `}` shall be replaced by *string2* . The parentheses or braces are optional if *string1* is a single character. The macro `$$` shall be replaced by the single character `$`.

Applications shall select macro names from the set of characters consisting solely 2
of periods, underscores, digits, and alphabetics from the portable character set 2
(see 2.4). A macro name shall not contain an equals-sign. Implementations may 2
allow other characters in macro names as extensions. 2

Macros can appear anywhere in the makefile. Macros in target lines shall be evaluated when the target line is read. Macros in command lines shall be evaluated when the command is executed. Macros in macro definition lines shall not be evaluated until the new macro being defined is used in a rule or command. A macro that has not been defined shall evaluate to a null string without causing any error condition.

The forms `$(`*string1* `[:`*subst1* `=[`*subst2* `]])` or `${`*string1* `[:`*subst1* `=[`*subst2* `]]}` can be used to replace all occurrences of *subst1* with *subst2* when the macro substitution is performed. The *subst1* to be replaced shall be recognized when it is a suffix at the end of a word in *string1* (where a ‘‘word,’’ in this context, is defined to be a string delimited by the beginning of the line, a `<blank>`, or a `<newline>`).

Macro assignments shall be accepted from the sources listed below, in the order shown. If a macro name already exists at the time it is being processed, the newer definition shall replace the existing definition.

(1) Macros defined in `make`'s built-in inference rules.

(2) The contents of the environment, including the variables with null values, in the order defined in the environment.

(3) Macros defined in the makefile(s), processed in the order specified.

(4) Macros specified on the command line. It is unspecified whether the internal macros defined in 6.2.7.7 are accepted from the command line.

If the `−e` option is specified, the order of processing sources (2) and (3) shall be reversed.

The `SHELL` macro shall be treated specially. It shall be provided by `make` and set to the pathname of the shell command language interpreter (see `sh` in 4.56). The **SHELL** environment variable shall not affect the value of the `SHELL` macro. If `SHELL` is defined in the makefile or is specified on the command line, it shall replace the original value of the `SHELL` macro, but shall not affect the **SHELL** environment variable. Other effects of defining `SHELL` in the makefile or on the command line are implementation defined.

6.2.7.5 Inference Rules

Inference rules are formatted as follows:

```
target :
<tab>command
[<tab>command ]
...
```

(*line that does not begin with* `<tab>` *or* #)

The *target* portion shall be a valid target name (see 6.2.7.3) and shall be of the form *.s2* or *.s1.s2* (where *.s1* and *.s2* are suffixes that have been given as prerequisites of the `.SUFFIXES` special target and *s1* and *s2* do not contain any slashes or periods.) If there is only one period in the target, it is a single-suffix inference rule. Targets with two periods are double-suffix inference rules. Inference rules can have only one target before the colon.

The makefile shall not specify prerequisites for inference rules; no characters other than white space shall follow the colon in the first line, except when creating the ‘‘empty rule,’’ described below. Prerequisites are inferred, as described below.

Inference rules can be redefined. A target that matches an existing inference rule shall overwrite the old inference rule. An ‘‘empty rule’’ can be created with a command consisting of simply a semicolon (that is, the rule still exists and is found during inference rule search, but since it is empty, execution has no effect). The empty rule also can be formatted as follows:

```
rule : ;
```

where zero or more <blank>s separate the colon and semicolon. 2

The `make` utility uses the suffixes of targets and their prerequisites to infer how a target can be made up-to-date. A list of inference rules defines the commands to be executed. By default, `make` contains a built-in set of inference rules. Additional rules can be specified in the makefile.

The special target `.SUFFIXES` contains as its prerequisites a list of suffixes that are to be used by the inference rules. The order in which the suffixes are specified 1 defines the order in which the inference rules for the suffixes are used. New suffixes shall be appended to the current list by specifying a `.SUFFIXES` special target in the makefile. A `.SUFFIXES` target with no prerequisites shall clear the list of suffixes. An empty `.SUFFIXES` target followed by a new `.SUFFIXES` list is required to change the order of the suffixes.

Normally, the user would provide an inference rule for each suffix. The inference 1 rule to update a target with a suffix *.s1* from a prerequisite with a suffix *.s2* is specified as a target *.s2.s1* . The internal macros provide the means to specify general inference rules. (See 6.2.7.7.) 1

When no target rule is found to update a target, the inference rules shall be checked. The suffix of the target (*.s1*) to be built is compared to the list of suffixes specified by the `.SUFFIXES` special targets. If the *.s1* suffix is found in `.SUFFIXES`, the inference rules shall be searched in the order defined for the first *.s2.s1* rule whose prerequisite file ($*.*s2*) exists. If the target is out-of-date with respect to this prerequisite, the commands for that inference rule shall be executed.

If the target to be built does not contain a suffix and there is no rule for the target, the single suffix inference rules shall be checked. The single-suffix inference rules define how to build a target if a file is found with a name that matches the 1 target name with one of the single suffixes appended. A rule with one suffix *.s2* is 1 the definition of how to build *target* from *target.s2* . The other suffix (*.s1*) is treated as null.

6.2.7.6 Libraries

If a target or prerequisite contains parentheses, it shall be treated as a member of an archive library. For the *lib*`(`*member*`.o)` expression *lib* refers to the name of the archive library and *member*`.o` to the member name. The member shall be an object file with the `.o` suffix. The modification time of the expression is the modification time for the member as kept in the archive library. See 6.1. The `.a` suffix refers to an archive library. The *.s2*`.a` rule is used to update a member in the library from a file with a suffix *.s2*.

6.2.7.7 Internal Macros

The `make` utility shall maintain five internal macros that can be used in target and inference rules. In order to clearly define the meaning of these macros, some clarification of the terms ''target rule,'' ''inference rule,'' ''target,'' and ''prerequisite'' is necessary.

Target rules are specified by the user in a makefile for a particular target. Inference rules are user- or `make`-specified rules for a particular class of target names. Explicit prerequisites are those prerequisites specified in a makefile on target lines. Implicit prerequisites are those prerequisites that are generated when inference rules are used. Inference rules are applied to implicit prerequisites or to explicit prerequisites that do not have target rules defined for them in the makefile. Target rules are applied to targets specified in the makefile.

Before any target in the makefile is updated, each of its prerequisites (both explicit and implicit) shall be updated. This shall be accomplished by recursively processing each prerequisite. Upon recursion, each prequisite shall become a target itself. Its prerequisites in turn shall be processed recursively until a target is found that has no prerequisites, at which point the recursion shall stop. The recursion then shall back up, updating each target as it goes.

In the definitions that follow, the word ''target'' refers to one of:

— A target specified in the makefile,

— An explicit prerequisite specified in the makefile that becomes the target when `make` processes it during recursion, or

— An implicit prerequisite that becomes a target when `make` processes it during recursion.

In the definitions that follow, the word ''prerequisite'' refers to either:

— An explicit prerequisite specified in the makefile for a particular target, or

— An implicit prerequisite generated as a result of locating an appropriate inference rule and corresponding file that matches the suffix of the target.

The five internal macros are:

`$@` The `$@` macro shall evaluate to the full target name of the current target, or the archive filename part of a library archive target. It shall be evaluated for both target and inference rules.

For example, in the `.c.a` inference rule, `$@` represents the out-of-date `.a` file to be built. Similarly, in a makefile target rule to build `lib.a` from `file.c`, `$@` represents the out-of-date `lib.a`.

`$%` The `$%` macro shall be evaluated only when the current target is an archive library member of the form *libname* `(`*member* `.o)`. In these cases, `$@` shall evaluate to *libname* and `$%` shall evaluate to *member* `.o`. The `$%` macro shall be evaluated for both target and inference rules.

For example, in a makefile target rule to build `lib.a(file.o)`, `$%` represents `file.o`—as opposed to `$@`, which represents `lib.a`.

`$?` The `$?` macro shall evaluate to the list of prerequisites that are newer than the current target. It shall be evaluated for both target and inference rules.

For example, in a makefile target rule to build `prog` from `file1.o`, `file2.o`, and `file3.o`, and where `prog` is not out of date with respect to `file1.o`, but is out of date with respect to `file2.o` and `file3.o`, `$?` represents `file2.o` and `file3.o`.

`$<` In an inference rule, `$<` shall evaluate to the file name whose existence allowed the inference rule to be chosen for the target. In the `.DEFAULT` rule, the `$<` macro shall evaluate to the current target name. The `$<` macro shall be evaluated only for inference rules.

For example, in the `.c.a` inference rule, `$<` represents the prerequisite `.c` file.

`$*` The `$*` macro shall evaluate to the current target name with its suffix deleted. It shall be evaluated at least for inference rules.

For example, in the `.c.a` inference rule, `$*.o` represents the out-of-date `.o` file that corresponds to the prerequisite `.c` file.

Each of the internal macros has an alternate form. When an uppercase `D` or `F` is appended to any of the macros, the meaning is changed to the *directory part* for `D` and *filename part* for `F`. The directory part is the path prefix of the file without a trailing slash; for the current directory, the directory part is `"."`. When the `$?` macro contains more than one prerequisite filename, the `$(?D)` and `$(?F)` [or `${?D}` and `${?F}`] macros expand to a list of directory name parts and filename parts respectively.

For the target *lib*`(`*member*`.o)` and the *.s2*`.a` rule, the internal macros are defined as:

`$<` *member*`.s2`

`$*` *member*

`$@` *lib*

`$?` *member*`.s2`

`$%` *member*`.o`

6.2.7.8 Default Rules

The default rules for `make` shall achieve results that are the same as if the following were used. Implementations that do not support the C Language Development Utilities Option may omit `CC`, `CFLAGS`, `YACC`, `YFLAGS`, `LEX`, `LFLAGS`, `LDFLAGS`, and the `.c`, `.y`, and `.l` inference rules. Implementations that do not support the FORTRAN Language Development Utilities Option may omit `FC`, `FFLAGS`, and the `.f` inference rules. Implementations may provide additional macros and rules.

NOTE: In a future version of this standard, the default rules may be specified separately from the make clause, such as with the language-dependent development options.

SUFFIXES AND MACROS

```
.SUFFIXES: .o .c .y .l .a .sh .f                                1

MAKE=make
AR=ar
ARFLAGS=-rv
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LDFLAGS=
CC=c89
CFLAGS=-O
FC=fort77
FFLAGS=-O 1
```

SINGLE SUFFIX RULES

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

.f:
	$(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<

.sh:
	cp $< $@
	chmod a+x $@
```

DOUBLE SUFFIX RULES

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

.f.o:
	$(FC) $(FFLAGS) -c $<

.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c                                           1
	mv y.tab.o $@

.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c                                          1
	mv lex.yy.o $@

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

.l.c:
	$(LEX) $(LFLAGS) $<
	mv lex.yy.c $@
```

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

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

6.2.8 Exit Status

When the −q option is specified, the make utility shall exit with one of the following values:

0	Successful completion.
1	The target was not up-to-date.
>1	An error occurred.

When the −q option is not specified, the make utility shall exit with one of the following values:

0	Successful completion.
>0	An error occurred.

6.2.9 Consequences of Errors

Default.

6.2.10 Rationale. *(This subclause is not a part of P1003.2)*

The make provided here is intended to provide the means for changing portable source code into runnable executables on a POSIX.2 system. It reflects the most common features present in System V and BSD makes.

Historically, the make utility has been an especially fertile ground for vendor- and research-organization-specific syntax modifications and extensions. Examples include:

- Syntax supporting parallel execution (Sequent, Cray, GNU, and others)
- Additional ‘‘operators’’ separating targets and their prerequisites (System V, BSD, and others)
- Specifying that command lines containing the strings ${MAKE} and $(MAKE) are executed when the −n option is specified (GNU and System V)
- Modifications of the meaning of internal macros when referencing libraries (BSD and others)

— Using a single instance of the shell for all of a target's command lines (BSD and others)

— Allowing spaces as well as tabs to delimit command lines (BSD)

— Adding C-preprocessor-style ''include'' and ''ifdef'' constructs (System V, GNU, BSD, and others)

— Remote execution of command lines (Sprite and others)

— Specifying additional special targets (Sun, BSD, System V, and most others).

Additionally, many vendors and research organizations have rethought the basic concepts of `make`, creating vastly extended, as well as completely new, syntaxes. Each of these versions of ''make'' fulfills the needs of a different community of users; it is unreasonable for this standard to require behavior that would be incompatible (and probably inferior) to existing practice for such a community.

In similar circumstances, when the industry has enough sufficiently incompatible formats as to make them irreconcilable, POSIX.2 has followed one or both of two courses of action. Commands have been renamed (`cksum`, `echo`, and `pax`) and/or command-line options have been provided to select the desired behavior (`grep`, `od`, and `pax`).

Because the syntax specified for the `make` utility is, by and large, a subset of the syntaxes accepted by almost all versions of `make`, it was decided that it would be counter-productive to change the name. And since the makefile itself is a basic unit of portability, it would not be completely effective to reserve a new option letter, such as `make -P`, to achieve the portable behavior. Therefore, the special target `.POSIX` was added to the makefile, allowing users to specify ''standard'' behavior. This special target does not preclude extensions in the `make` utility, or such extensions being used by the makefile specifying the target; it does, however, preclude any extensions from being applied that could alter the behavior of previously valid syntax; such extensions must be controlled via command-line options or new special targets. It is incumbent upon portable makefiles to specify the `.POSIX` special target in order to guarantee that they are not affected by local extensions.

The portable version of `make` described in this clause is not intended to be the state of the art software generation tool and, as such, some newer and more leading-edge features have not been included. An attempt has been made to describe the portable makefile in a manner that does not preclude such extensions as long as they do not disturb the portable behavior described here.

One use of this `make` and the makefile syntax is as a format that newer versions of `make` can generate for portability purposes.

Examples, Usage

The following command:

```
make
```

makes the first target found in the makefile.

The following command:

```
make junk
```

makes the target `junk`.

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
        c89 a.o b.o –o pgm

a.o: incl.h a.c
        c89 –c a.c

b.o: incl.h b.c
        c89 –c b.c
```

An example for making optimized `.o` files from `.c` files is:

```
.c.o:
        c89 –c –O $*.c
```

or:

```
.c.o:
        c89 –c –O $<
```

The most common use of the archive interface follows. Here, it is assumed that the source files are all C language source:

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

The `.c.a` rule is used to make `file1.o`, `file2.o`, and `file3.o` and insert them 1
into `lib`. 1

The `–k` and `–S` options are both present so that the relationship between the command line, the **MAKEFLAGS** variable, and the makefile can be controlled precisely. If the `k` flag is passed in **MAKEFLAGS** and a command is of the form:

```
$(MAKE) -S foo
```

then the default behavior is restored for the child `make`.

When the `–n` option is specified, it is always added to **MAKEFLAGS** . This allows a recursive `make -n` *target* to be used to see all of the action that would be taken to update *target* .

The definition of **MAKEFLAGS** allows both the System V letter string and the BSD command-line formats. The two formats are sufficiently different to allow implementations to support both without ambiguity.

Because of widespread historical practice, interpreting a # number sign inside a variable as the start of a comment has the unfortunate side effect of making it impossible to place a number sign in a variable, thus forbidding something like

```
CFLAGS = "-D COMMENT_CHAR='#'"
```

Earlier drafts stated that an ‘‘unquoted’’ number sign was treated as the start of a comment. The `make` utility does not pay any attention to quotes. A number sign starts a comment regardless of its surroundings.

The treatment of escaped `<newline>`s throughout the makefile is historical practice. For example, the inference rule:

```
.c.o\
:
```

works and the macro

```
f=      bar baz\

        biz
a:
        echo ==$f==
```

will echo `==bar baz biz==`.

If `$?` were

```
/usr/include/stdio.h /usr/include/unistd.h foo.h
```

then `$(?D)` would be

```
/usr/include /usr/include .
```

and `$(?F)` would be

```
stdio.h unistd.h foo.h
```

The contents of the built-in rules can be viewed by running:

```
make -p -f /dev/null 2>/dev/null
```

Many historical `make`s stop chaining together inference rules when an intermedi- 1
ate target is nonexistent. For example, it might be possible for a `make` to deter- 1
mine that both `.y.c` and `.c.o` could be used to convert a `.y` to a `.o`. Instead, in 1
this case, `make` requires the use of a `.y.o` rule. 1

The text about ‘‘other implementation-defined pathnames may also be tried’’ in addition to `./makefile` and `./Makefile` is to allow such extensions as `SCCS/s.Makefile` and other variations. It was made an implementation-defined requirement (as opposed to unspecified behavior) to highlight surprising implementations that might select something unexpected like `/etc/Makefile`.

For inference rules, the description of `$<` and `$?` seem similar. However, an example shows the minor difference. In a makefile containing

```
foo.o: foo.h
```

if `foo.h` is newer than `foo.o`, yet `foo.c` is older than `foo.o`, the built-in rule to make `foo.o` from `foo.c` will be used, with `$<` equal to `foo.c` and `$?` equal to `foo.h`. (If `foo.c` is also newer than `foo.o`, `$<` is equal to `foo.c` and `$?` is equal to ‘‘`foo.h foo.c`’’.)

History of Decisions Made

Earlier drafts contained the macro `NPROC` as a means of specifying that `make` should use *n* processes to do the work required. While this feature is a valuable extension for many systems, it is not common usage and could require other nontrivial extensions to makefile syntax. This extension is not required by the standard, but could be provided as a compatible extension. The macro `PARALLEL` is used by some historical systems with essentially the same meaning (but without using a name that is a common system limit value). It is suggested that implementors recognize the existing use of `NPROC` and/or `PARALLEL` as extensions to `make`.

The default rules are based on System V. The default `CC=` value is `c89` instead of `cc` because POSIX.2 does not standardize the utility named `cc`. Thus, every conforming application would be required to define `CC=c89` to expect to run. There is no advantage conferred by the hope that the makefile might hit the ''preferred'' compiler because there is no way that this can be guaranteed to work. Also, since the portable makescript can only use the `c89` options, no advantage is conferred in terms of what the script can do. It is a quality of implementation issue as to whether `c89` is as good as `cc`.

Since SCCS and RCS are not part of POSIX.2, all `make` references to SCCS extensions have been omitted.

The `−d` option to `make` is frequently used to produce debugging information, but is too implementation-dependent to add to the standard.

The `−p` option is not passed in **MAKEFLAGS** on most existing implementations and to change this would cause many implementations to break without sufficiently increased portability.

Commands that begin with a plus-sign (+) are executed even if the `−n` option is present. Based on the GNU version of `make`, the behavior of `−n` when the plus-sign prefix is encountered has been extended to apply to `−q` and `−t` as well. However, the System V convention of forcing command execution with `−n` when a target's command line contains either of the strings `$(MAKE)` or `${MAKE}` has not been adopted. This functionality appeared in earlier drafts, but the danger of this approach was pointed out with the following example of a portion of a makefile:

```
subdir:
        cd subdir; rm all_the_files; $(MAKE)
```

The loss of the System V behavior in this case is well-balanced by the safety afforded to other makefiles that were not aware of this situation. In any event, the command-line plus-sign prefix can provide the desired functionality.

The double colon in the target rule format is supported in BSD systems to allow more than one target line containing the same target name to have commands associated with it. Since this is not functionality described in the *SVID* or XPG3, it has been allowed as an extension, but not mandated.

The default rules are provided with text specifying that the built-in rules are to be the same *as if* the listed set were used. The intent is that implementations should be able to use the rules without change, but will be allowed to alter them in ways that do not affect the primary behavior.

The best way to provide portable makefiles is to include all of the rules needed in the makefile itself. The rules provided use only features provided by other parts of the standard. The default rules include rules for optional commands in the standard. Only rules pertaining to commands that are provided are needed in an implementation's default set.

The argument could be made to drop the default rules list from the standard. They provide convenience, but do not enhance portability of applications. The prime benefit is in portability of users who wish to type `make command` and have the command build from a `command.c` file.

The historical **MAKESHELL** feature was omitted. In some implementations it is used to provide a way of letting a user override the shell to be used to run `make` commands. This was confusing; for a portable `make`, the shell should be chosen by the makefile writer or specified on the `make` command line and not by a user running `make`.

The `make` utilities in most historical implementations process the prerequisites of a target in left-to-right order, and the POSIX.2 makefile format requires this. It supports the standard idiom used in many makefiles that produce `yacc` programs, for example:

```
foo:    y.tab.o lex.o main.o
        $(CC) $(CFLAGS) -o $@ t.tab.o lex.o main.o
```

In this example, if `make` chose any arbitrary order, the `lex.o` might not be made with the correct `y.tab.h`. Although there may be better ways to express this relationship, it is widely used historically. Implementations that desire to update prerequisites in parallel should require an explicit extension to `make` or the makefile format to accomplish it, as described previously.

The algorithm for determining a new entry for target rules is partially unspecified. Some historical `make`s allow blank, empty, or comment lines within the collection of commands marked by leading `<tab>`s. A conforming makefile must ensure that each command starts with a `<tab>`, but implementations are free to ignore blank, empty, and comment lines without triggering the start of a new entry.

The Asynchronous Events subclause includes having SIGTERM and SIGHUP, along with the more traditional SIGINT and SIGQUIT, remove the current target unless directed not to. SIGTERM and SIGHUP were added to parallel other utilities that have historically cleaned up their work as a result of these signals. All but SIGQUIT is required to resend itself the signal it received to cause `make` to exit with a status that reflects the signal. The results from SIGQUIT are partially unspecified because, on systems that create `core` files upon receipt of SIGQUIT, the `core` from `make` would conflict with a core file from the command that was running when the SIGQUIT arrived. The main concern here was to prevent damaged files from appearing up-to-date when `make` is rerun.

The `.PRECIOUS` special target was extended to globally affect all targets (by specifying no prerequisites). The `.IGNORE` and `.SILENT` special targets were extended to allow prerequisites; it was judged to be more useful in some cases to be able to turn off errors or echoing for a list of targets than for the entire makefile. These extensions to System V's `make` were made to match historical practice from the BSD `make`.

Macros are not exported to the environment of commands to be run. This was never the case in any historical `make` and would have serious consequences. The environment is the same as the environment to `make` except that **MAKEFLAGS** and macros defined on the `make` command line are added.

Some implementations do not use *system* () for all command lines, as required by the POSIX.2 portable makefile format; as a performance enhancement, they select lines without shell metacharacters for direct execution by *execve* (). There is no requirement that *system* () be used specifically, but merely that the same results be achieved. The metacharacters typically used to bypass the direct *execve* () execution have been any of:

```
= | ^ ( ) ; & < > * ? [ ] : $ ‘ ’ " \ \n
```

The default in some advanced versions of `make` is to group all the command lines for a target and execute them using a single shell invocation; the System V method is to pass each line individually to a separate shell. The single-shell method has the advantages in performance and the lack of a requirement for many continued lines. However, converting to this newer method has caused portability problems with many historical makefiles, so the behavior with the POSIX makefile is specified to be the same as System V's. It is suggested that the special target `.ONESHELL` be used as an implementation extension to achieve the single-shell grouping for a target or group of targets.

Novice users of `make` have had difficulty with the historical need to start commands with a `<tab>` character. Since it is often difficult to discern differences between `<tab>` and `<space>` characters on terminals or printed listings, confusing bugs can arise. In earlier drafts, an attempt was made to correct this problem by allowing leading `<blank>`s instead of `<tab>`s. However, implementors reported many makefiles that failed in subtle ways following this change and it is difficult to implement a `make` that unambiguously can differentiate between macro and command lines. There is extensive historical practice of allowing leading spaces before macro definitions. Forcing macro lines into column 1 would be a significant backward compatibility problem for some makefiles. Therefore, historical practice was restored.

The System V INCLUDE feature was considered, but not included. This would treat a line that began in the first column and contained INCLUDE *<filename>* as an indication to read *<filename>* at that point in the makefile. This is difficult to use in a portable way and it raises concerns about nesting levels and diagnostics. System V, BSD, GNU, and others have used different methods for including files.

Macros used within other macros are evaluated when the new macro is used rather than when the new macro is defined. Therefore:

```
MACRO = value1
NEW   = $(MACRO)
MACRO = value2

target:
        echo $(NEW)
```

would produce *value2* and not *value1* since NEW was not expanded until it was needed in the echo command line.

The System V dynamic dependency feature was not added. It would support:

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

that would expand to

```
cat: cat.c
```

This feature exists only in the new version of System V make and, while useful, is not in wide usage. This means that macros are expanded twice for prerequisites: once at makefile parse time and once at target update time.

Consideration was given to adding metarules to the POSIX make. This would make "%.o: %.c" the same as ".c.o:". This is quite useful and available from some vendors, but it would cause too many changes to this make to support. It would have introduced rule chaining and new substitution rules. However, the rules for target names have been set to reserve the % and " characters. These are traditionally used to implement metarules and quoting of target names, respectively. Implementors are strongly encouraged to use these characters only for these purposes.

A request was made to extend the suffix delimiter character from a period to any character. The metarules in newer makes solves this problem in a more general way. POSIX.2 is staying with the more conservative historical definition until a clear industry consensus on make technology might prompt a revision of this standard.

The standard output format for the −p option is not described because it is primarily a debugging option and the format is not generally useful to programs. In historical implementations the output is not suitable for use in generating makefiles. The −p format has been variable across historical implementations. Therefore, the definition of −p was only to provide a consistently named option for obtaining make script debugging information.

Some historical implementations have not cleared the suffix list with −r.

Implementations should be aware that some historical applications have intermixed *target_name* and *macro*=*name* operands on the command line, expecting that all of the macros will be processed before any of the targets are dealt with. Portable applications do not do this, but some backward compatibility support may be warranted.

Empty inference rules are specified with a semicolon command rather than omitting all commands, as described in a previous draft. The latter case has no traditional meaning and is reserved for implementation extensions, such as in GNU make.

6.3 strip — Remove unnecessary information from executable files

6.3.1 Synopsis

strip *file* ...

6.3.2 Description

The strip utility shall remove from executable files named by the *file* operands any information the implementor deems unnecessary to proper execution of those files. The nature of that information is unspecified. The effect of strip shall be the same as the use of the −s option to any of the compilers defined by this standard.

6.3.3 Options

None.

6.3.4 Operands

The following operand shall be supported by the implementation:

file A pathname referring to an executable file.

6.3.5 External Influences

6.3.5.1 Standard Input

None.

6.3.5.2 Input Files

The input files shall be in the form of executable files successfully produced by any compiler defined by this standard.

6.3.5.3 Environment Variables

The following environment variables shall affect the execution of strip: