IEEE P1003.2 Draft 11.2 – September 1991

Copyright © 1991 by the
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities only, and subject to the restrictions contained herein.

Permission is hereby also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position, subject to the restrictions contained herein.

Permission is hereby also granted to the preceding entities to make limited copies of this document in an electronic form only for the stated activities.

The following restrictions apply to reproducing or transmitting the document in any form: 1) all copies or portions thereof must identify the document's IEEE project number and draft number, and must be accompanied by this entire notice in a prominent location; 2) no portion of this document may be redistributed in any modified or abridged form without the prior approval of the IEEE Standards Department.

Other entities seeking permission to reproduce this document, or any portion thereof, for standardization or other activities, must contact the IEEE Standards Department for the appropriate license.

Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

505 command shall not be written to standard output before it is executed.

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

508 #### 6.2.7.3 Target Rules

509 Target rules are formatted as follows:

510 *target* [*target* ...]: [*prerequisite* ...][;*command*] 1
511 [<tab>*command* 1
512 <tab>*command* 1
513 ...] 1

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

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

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

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

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

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

535 .IGNORE Prerequisites of this special target are targets themselves; this
536 shall cause errors from commands associated with them to be
537 ignored in the same manner as specified by the –i option. Subse-
538 quent occurrences of .IGNORE shall add to the list of targets ignor-
539 ing command errors. If no prerequisites are specified, make shall
540 behave as if the –i option had been specified and errors from all
541 commands associated with all targets shall be ignored.

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

546 `.PRECIOUS` Prerequisites of this special target shall not be removed if `make`
547 receives one of the asynchronous events explicitly described in
548 6.2.5.4. Subsequent occurrences of `.PRECIOUS` shall add to the
549 list of precious files. If no prerequisites are specified, all targets
550 in the makefile shall be treated as if specified with `.PRECIOUS`.

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

559 `.SUFFIXES` Prerequisites of `.SUFFIXES` shall be appended to the list of known
560 suffixes and are used in conjunction with the inference rules (see
561 6.2.7.5). If `.SUFFIXES` does not have any prerequisites, the list of
562 known suffixes shall be cleared. Makefiles shall not associate
563 commands with `.SUFFIXES`.

564 Targets with names consisting of a leading period followed by the uppercase
565 letters `POSIX` and then any other characters are reserved for future standardiza-
566 tion. Targets with names consisting of a leading period followed by one or more
567 uppercase letters are reserved for implementation extensions.

568 ### 6.2.7.4 Macros

569 Macro definitions are in the form:

570 *string1* `=` [*string2*] 1

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

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

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

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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.

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 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 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.)

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 target name with one of the single suffixes appended. A rule with one suffix *.s2* is 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*.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

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
	mv y.tab.o $@

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

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

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
.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)

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

— 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
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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 into `lib`.

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
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 makes stop chaining together inference rules when an intermediate target is nonexistent. For example, it might be possible for a make to determine that both .y.c and .c.o could be used to convert a .y to a .o. Instead, in this case, make requires the use of a .y.o rule.

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’’.)

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

946 **History of Decisions Made**

947 Earlier drafts contained the macro `NPROC` as a means of specifying that `make`
948 should use *n* processes to do the work required. While this feature is a valuable
949 extension for many systems, it is not common usage and could require other non-
950 trivial extensions to makefile syntax. This extension is not required by the stan-
951 dard, but could be provided as a compatible extension. The macro `PARALLEL` is 1
952 used by some historical systems with essentially the same meaning (but without 1
953 using a name that is a common system limit value). It is suggested that imple- 1
954 mentors recognize the existing use of `NPROC` and/or `PARALLEL` as extensions to 1
955 `make`. 1

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

964 Since SCCS and RCS are not part of POSIX. 2, all `make` references to SCCS exten-
965 sions have been omitted.

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

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

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

978 ```
979 subdir:
        cd subdir; rm all_the_files; $(MAKE)
```

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

983 The double colon in the target rule format is supported in BSD systems to allow
984 more than one target line containing the same target name to have commands
985 associated with it. Since this is not functionality described in the *SVID* or XPG 3, it
986 has been allowed as an extension, but not mandated.

987 The default rules are provided with text specifying that the built-in rules are to be
988 the same *as if* the listed set were used. The intent is that implementations

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.
```

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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1118 `make`.

1119 6.3 `strip` — Remove unnecessary information from executable 1120 files

1121 6.3.1 Synopsis

1122 `strip` *file* ...

1123 6.3.2 Description

1124 The `strip` utility shall remove from executable files named by the *file* operands
1125 any information the implementor deems unnecessary to proper execution of those
1126 files. The nature of that information is unspecified. The effect of `strip` shall be
1127 the same as the use of the `-s` option to any of the compilers defined by this stan-
1128 dard.

1129 6.3.3 Options

1130 None.

1131 6.3.4 Operands

1132 The following operand shall be supported by the implementation:

1133 *file* A pathname referring to an executable file.

1134 6.3.5 External Influences

1135 6.3.5.1 Standard Input

1136 None.

1137 6.3.5.2 Input Files

1138 The input files shall be in the form of executable files successfully produced by
1139 any compiler defined by this standard.

1140 6.3.5.3 Environment Variables

1141 The following environment variables shall affect the execution of `strip`:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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).

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

6.3.5.4 Asynchronous Events

Default.

6.3.6 External Effects

6.3.6.1 Standard Output

None.

6.3.6.2 Standard Error

Used only for diagnostic messages.

6.3.6.3 Output Files

The `strip` utility shall produce executable files of unspecified format.

6.3.7 Extended Description

None.

6.3.8 Exit Status

The `strip` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

6.3.9 Consequences of Errors

Default.

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

Examples, Usage

None.

History of Decisions Made

Historically, this utility has been used to remove the symbol table from an executable file. It was included since it is known that the amount of symbolic information can amount to several megabytes; the ability to remove it in a portable manner was deemed important, especially for smaller systems.

The behavior of `strip` is said to be the same as the `-s` option to a compiler. While the end result is essentially the same it is not required to be identical. The same effect can be achieved with either `-s` during a compile or a `strip` on the final object file.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 7: Language-Independent System Services

This clause contains functional specifications for services that give applications access to features defined elsewhere in this standard. These services allow applications written in high-level languages to

(1) execute commands using the shell language,

(2) obtain values of environment variables,

(3) perform regular expression and pattern matching,

(4) process command arguments in a standard manner,

(5) generate pathnames from a pattern,

(6) perform shell word expansions,

(7) obtain system configuration information, and

(8) set locale control information

This clause does not define interfaces, but services that shall be provided by the interfaces in a language-dependent binding. This clause is optional, in that an implementation is not required to support any language binding to these services. However, any language binding shall support all of the services described here. Implementations therefore provide support for services in this clause by supplying a language-dependent binding such as the one defined in Annex B. Such a system would specify conformance to the language-dependent binding, not to the language-independent bindings given here.

7.0.1 Language-Independent System Services Rationale. *(This subclause is not a part of P1003.2)*

Section 7 essentially is a metastandard, in that it specifies services that must be in a language-dependent binding. An implementation conforms to a specific language-dependent binding such as for the C language, in Annex B, and the language-dependent binding must conform to the specifications in this clause.

In this standard, the language-independent specifications have not yet been developed. The language-independent syntax is being created in parallel by the POSIX. 1 working group. Therefore, the C language bindings temporarily described in Annex B are actually the full interface specifications. It is the intention of the P1003.2 working group to rectify this situation in a later supplement by moving the majority of the interface specifications back into this clause,

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

leaving Annex B with only brief descriptions of the C bindings to those services.

This clause does not attempt to include everything that would be required of a language binding. The services here are those that are necessary to make use of features defined elsewhere in the standard, but that are not normally available in every language. Clearly a language that could not open, read, and write the files manipulated by the utilities in this standard would not be very useful, but this service is normally provided by any language and therefore isn't called out here. The ability to obtain values of environment variables exported from the shell, on the other hand, is not universally available, so that service is included here.

7.1 Shell Command Interface

7.1.1 Execute Shell Command

Any language binding to Language-Independent System Services shall include a facility to execute a shell command.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *system* () function described in B.3.1.

7.1.2 Pipe Communications with Programs

Any language binding to Language-Independent System Services shall include a facility to execute a shell command, and to write the standard input or read the standard output of that command via a pipe.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *popen* () and *pclose* () functions described in B.3.2.

7.2 Access Environment Variables

Any language binding to Language-Independent System Services shall include a facility to obtain values of environment variables, as specified in POSIX .1 {8}.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *getenv* () function described in POSIX .1 {8} 4.6.1.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.