Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'make.1p' command

$ man make.1p

MAKE(1P) POSIX Programmer's Manual MAKE(1P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

make ? maintain, update, and regenerate groups of programs (DEVELOP? MENT)

SYNOPSIS

make [-einpqrst] [-f makefile]... [-k|-S] [macro=value...] [target_name...]

DESCRIPTION

The make utility shall update files that are derived from other files. A typical case is one where object files are derived from the corre? sponding source files. The make utility examines time relationships and shall update those derived files (called targets) that have modified times earlier than the modified times of the files (called prerequi? sites) from which they are derived. A description file (makefile) con? tains a description of the relationships between files, and the com? mands that need to be executed to update the targets to reflect changes in their prerequisites. Each specification, or rule, shall 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 rule:

1. Inference rules, which have one target name with at least one <pe? riod> ('.') and no <slash> ('/')
2. Target rules, which can have more than one target name

In addition, make shall have a collection of built-in macros and infer? ence rules that infer prerequisite relationships to simplify mainte? nance of programs.

To receive exactly the behavior described in this section, the user shall ensure that a portable makefile shall:

* Include the special target .POSIX
* Omit any special target reserved for implementations (a leading pe? riod followed by uppercase letters) that has not been specified by this section

The behavior of make is unspecified if either or both of these condi? tions are not met.

OPTIONS

The make utility shall conform to the Base Definitions volume of POSIX.1?2017, Section 12.2, Utility Syntax Guidelines, except for Guideline 9.

The following options shall be supported:

-e Cause environment variables, including those with null val? ues, 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 unspec? ified.

-i Ignore error codes returned by invoked commands. This mode is the same as if the special target .IGNORE were specified

without prerequisites.

-k Continue to update other targets that do not depend on the current target if a non-ignored 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 defini? tions and target descriptions. The output format is unspeci? fied.

-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 makefile com? mand 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 com? mands to bring a target up-to-date. This shall be the default and the opposite of -k.

-s Do not write makefile 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.

-t Update the modification time of each target as though a touch target had been executed. Targets that have prerequisites but no commands (see Target Rules), 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 makefile com? mand lines associated with each target are not executed. How? ever, a command line with a <plus-sign> ('+') prefix shall be executed.

Any options specified in the MAKEFLAGS environment variable shall be evaluated before any options specified on the make utility command line. If the -k and -S options are both specified on the make utility command line or by the MAKEFLAGS environment variable, the last option specified shall take precedence. If the -f or -p options appear in the MAKEFLAGS environment variable, the result is undefined.

OPERANDS

The following operands shall be supported:

target_name

Target names, as defined in the EXTENDED DESCRIPTION section. If no target is specified, while make is processing the make? files, the first target that make encounters that is not a special target or an inference rule shall be used.

macro=value

Macro definitions, as defined in Macros.

If the target_name and macro=value operands are intermixed on the make utility command line, the results are unspecified.

STDIN

The standard input shall be used only if the makefile option-argument is '-'. See the INPUT FILES section.

INPUT FILES

The input file, otherwise known as the makefile, is a text file con? taining rules, macro definitions, include lines, and comments. See the EXTENDED DESCRIPTION section.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of make:

LANG Provide a default value for the internationalization vari? ables that are unset or null. (See the Base Definitions vol? ume of POSIX.1?2017, Section 8.2, Internationalization Vari? ables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard er? ror.

MAKEFLAGS

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

* The characters are option letters without the leading <hyphen-minus> characters or <blank> separation used on a make utility command line.
* The characters are formatted in a manner similar to a portion of the make utility command line: options are preceded by <hyphen-minus> characters and <blank>-sepa? rated as described in the Base Definitions volume of POSIX.1?2017, Section 12.2, Utility Syntax Guidelines. The macro=value macro definition operands can also be in? cluded. The difference between the contents of MAKEFLAGS and the make utility command line is that the contents of the variable shall not be subjected to the word expan? sions (see Section 2.6, Word Expansions) associated with parsing the command line values.

NLSPATH Determine the location of message catalogs for the processing of LC_MESSAGES.

PROJECTDIR

Provide a directory to be used to search for SCCS files not found in the current directory. In all of the following cases, the search for SCCS files is made in the directory

SCCS in the identified directory. If the value of PROJECTDIR begins with a <slash>, it shall be considered an absolute pathname; otherwise, the value of PROJECTDIR is treated as a user name and that user's initial working directory shall be examined for a subdirectory src or source. If such a direc? tory is found, it shall be used. Otherwise, the value is used as a relative pathname.

If PROJECTDIR is not set or has a null value, the search for SCCS files shall be made in the directory SCCS in the current directory.

The setting of PROJECTDIR affects all files listed in the re? mainder of this utility description for files with a compo? nent named SCCS.

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 make? file or on the command line. All other environment variables, including those with null values, shall be used as macros, as defined in Macros.

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 un? less one of the -n, -p, or -q options was specified. Any targets re? moved in this manner shall be reported in diagnostic messages of un? specified format, written to standard error. After this cleanup process, if any, make shall take the standard action for all other sig? nals.

STDOUT

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 out? put indicating that no action was taken. If the -t option is present

and a file is touched, make shall write to standard output a message of unspecified format indicating that the file was touched, including the filename of the file.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Files can be created when the -t option is present. Additional files can also be created by the utilities invoked by make.

EXTENDED DESCRIPTION

The make utility attempts to perform the actions required to ensure that the specified targets are up-to-date. A target shall be considered up-to-date if it exists and is newer than all of its dependencies, or if it has already been made up-to-date by the current invocation of make (regardless of the target's existence or age). A target may also be considered up-to-date if it exists, is the same age as one or more of its prerequisites, and is newer than the remaining prerequisites (if any). The make utility shall treat all prerequisites as targets them? selves 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 whether the corre? sponding targets are out-of-date.

To ensure that a target is up-to-date, make shall ensure that all of the prerequisites of a target are up-to-date, then check to see if the target itself is up-to-date. If the target is not up-to-date, the tar? get shall be made up-to-date by executing the rule's commands (if any). If the target does not exist after the target has been successfully made up-to-date, the target shall be treated as being newer than any target for which it is a prerequisite.

If a target exists and there is neither a target rule nor an inference rule for the target, the target shall be considered up-to-date. It shall be an error if make attempts to ensure that a target is up-to-date but the target does not exist and there is neither a target rule nor an inference rule for the target.

Makefile Syntax

A makefile can contain rules, macro definitions (see Macros), include lines, and comments. There are two kinds of rules: inference rules and target rules. The make utility shall contain a set of built-in infer? ence 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 is defined more than once, the value of the rule shall be that of the last one specified.

Macros can also be defined more than once, and the value of the macro is specified in Macros. There are three kinds of comments: blank lines, empty lines, and a <number-sign> ('#') and all following charac? ters up to the first unescaped <newline> character. Blank lines, empty lines, and lines with <number-sign> ('#') as the first character on the line are also known as comment lines.

By default, the following files shall be tried in sequence: ./makefile and ./Makefile. If neither ./makefile or ./Makefile are found, other implementation-defined files may also be tried. On XSI-conformant sys? tems, the additional files ./s.makefile, SCCS/s.makefile, ./s.Makefile, and SCCS/s.Makefile shall also be tried.

The -f option shall direct make to ignore any of these default files and use the specified argument as a makefile instead. If the '-' argu? ment is specified, standard input shall be used.

The term makefile is used to refer to any rules provided by the user, whether in ./makefile or its variants, or specified by the -f option.

The rules in makefiles shall consist of the following types of lines: target rules, including special targets (see Target Rules), inference rules (see Inference Rules), macro definitions (see Macros), and com? ments.

Target and Inference Rules may contain command lines. Command lines can have a prefix that shall be removed before execution (see Makefile Execution).

When an escaped <newline> (one preceded by a <backslash>) is found any? where in the makefile except in a command line, an include line, or a

line immediately preceding an include line, it shall be replaced, along with any leading white space on the following line, with a single <space>. When an escaped <newline> is found in a command line in a makefile, the command line shall contain the <backslash>, the <new? line>, and the next line, except that the first character of the next line shall not be included if it is a <tab>. When an escaped <newline> is found in an include line or in a line immediately preceding an in? clude line, the behavior is unspecified.

Include Lines

If the word include appears at the beginning of a line and is followed by one or more <blank> characters, the string formed by the remainder of the line shall be processed as follows to produce a pathname:

* The trailing <newline>, any <blank> characters immediately preced? ing a comment, and any comment shall be discarded. If the resulting string contains any double-quote characters ("") the behavior is unspecified.
* The resulting string shall be processed for macro expansion (see Macros).
* Any <blank> characters that appear after the first non-<blank> shall be used as separators to divide the macro-expanded string into fields. It is unspecified whether any other white-space char? acters are also used as separators. It is unspecified whether path? name expansion (see Section 2.13, Pattern Matching Notation) is also performed.
* If the processing of separators and optional pathname expansion re? sults in either zero or two or more non-empty fields, the behavior is unspecified. If it results in one non-empty field, that field is taken as the pathname.

If the pathname does not begin with a '/' it shall be treated as rela? tive to the current working directory of the process, not relative to the directory containing the makefile. If the file does not exist in this location, it is unspecified whether additional directories are searched.

The contents of the file specified by the pathname shall be read and processed as if they appeared in the makefile in place of the include line. If the file ends with an escaped <newline> the behavior is un? specified.

The file may itself contain further include lines. Implementations shall support nesting of include files up to a depth of at least 16.

Makefile Execution

Makefile command lines shall be processed one at a time.

Makefile command lines can have one or more of the following prefixes: a <hyphen-minus> ('-'), an at-sign ('@'), or a <plus-sign> ('+'). These shall modify the way in which make processes the command.

- If the command prefix contains a <hyphen-minus>, 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 make utility 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 makefile command line that shall be executed even if -n, -q, or -t is specified.

An execution line is built from the command line by removing any prefix characters. Except as described under the at-sign prefix, the execution line shall be written to the standard output, optionally preceded by a <tab>. The execution line shall then be executed by a shell as if it were passed as the argument to the system() interface, except that if errors are not being ignored then the shell -e option shall also be in effect. If errors are being ignored for the command (as a result of the -i option, a '-' command prefix, or a .IGNORE special target), the shell -e option shall not be in effect. The environment for the command being executed shall contain all of the variables in the environment of

make.

By default, when make receives a non-zero status from the execution of a command, it shall terminate with an error message to standard error.

Target Rules

Target rules are formatted as follows:

target [target...]: [prerequisite...][;command]

[<tab>command

<tab>command

...]

line that does not begin with <tab>

Target entries are specified by a <blank>-separated, non-null list of targets, then a <colon>, then a <blank>-separated, possibly empty list of prerequisites. Text following a <semicolon>, if any, and all follow? ing lines that begin with a <tab>, are makefile command lines to be ex? ecuted to update the target. The first non-empty line that does not be? gin with a <tab> or '#' shall begin a new entry. Any comment line may begin a new entry.

Applications shall select target names from the set of characters con? sisting solely of periods, underscores, digits, and alphabetics from the portable character set (see the Base Definitions volume of POSIX.1?2017, Section 6.1, Portable Character Set). Implementations may allow other characters in target names as extensions. The interpre? tation of targets containing the characters '%' and "" is implementa? tion-defined.

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, the application shall ensure that it is 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 The application shall ensure that this special target is specified without prerequisites or commands. If it appears as the first non-comment line in the makefile, make shall process the makefile as specified by this section; 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 de? scribed in the ASYNCHRONOUS EVENTS section. Subsequent occur? rences of .PRECIOUS shall add to the list of precious files. If no prerequisites are specified, all targets in the make? file shall be treated as if specified with .PRECIOUS.

.SCCS_GET The application shall ensure that this special target is specified without prerequisites. If this special target is included in a makefile, the commands specified with this tar? get shall replace the default commands associated with this special target (see Default Rules). The commands specified with this target are used to get all SCCS files that are not found in the current directory.

When source files are named in a dependency list, make shall treat them just like any other target. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile. When a target has no dependencies, but is present in the directory, make shall assume that that file is up-to-date. If, however, an SCCS file named SCCS/s.source_file is found for a target

source_file, make compares the timestamp of the target file with that of the SCCS/s.source_file to ensure the target is up-to-date. If the target is missing, or if the SCCS file is newer, make shall automatically issue the commands specified for the .SCCS_GET special target to retrieve the most recent version. However, if the target is writable by anyone, make shall not retrieve a new version.

.SILENT Prerequisites of this special target are targets themselves; this shall cause commands associated with them not to be written to the standard output before they are executed. Sub? sequent occurrences of .SILENT shall add to the list of tar? gets 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 Inference Rules). If .SUFFIXES does not have any prerequisites, the list of known suffixes shall be cleared.

The special targets .IGNORE, .POSIX, .PRECIOUS, .SILENT, and .SUFFIXES shall be specified without commands.

Targets with names consisting of a leading <period> followed by the up? percase 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 im? plementation extensions.

Macros

Macro definitions are in the form:

string1 = [string2]

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 character ('#') or an unescaped <newline>. Any <blank> characters immediately before or after the <equals-sign> shall

be ignored.

Applications shall select macro names from the set of characters con? sisting solely of periods, underscores, digits, and alphabetics from the portable character set (see the Base Definitions volume of POSIX.1?2017, Section 6.1, Portable Character Set). A macro name shall not contain an <equals-sign>. Implementations may allow other charac? ters in macro names as extensions.

Macros can appear anywhere in the makefile. Macro expansions using the forms $(string1) or ${string1} shall be replaced by string2, as fol? lows:

* Macros in target lines shall be evaluated when the target line is read.
* Macros in makefile command lines shall be evaluated when the com? mand is executed.
* Macros in the string before the <equals-sign> in a macro definition shall be evaluated when the macro assignment is made.
* Macros after the <equals-sign> in a macro definition shall not be evaluated until the defined macro is used in a rule or command, or before the <equals-sign> in a macro definition.

The parentheses or braces are optional if string1 is a single charac? ter. The macro $$ shall be replaced by the single character '$'. If string1 in a macro expansion contains a macro expansion, the results are unspecified.

Macro expansions using 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 <new? line>). If string1 in a macro expansion contains a macro expansion, the results are unspecified. If a <percent-sign> character appears as part of subst1 or subst2 after any macros have been recursively ex? panded, the results are unspecified.

Macro expansions in string1 of macro definition lines shall be evalu?
ated when read. Macro expansions in string2 of macro definition lines
shall be performed when the macro identified by string1 is expanded in
a rule or command.

Macro definitions shall be taken from the following sources, in the
following logical order, before the makefile(s) are read.

1. Macros specified on the make utility command line, in the order
   specified on the command line. It is unspecified whether the inter?
   nal macros defined in Internal Macros are accepted from this
   source.
2. Macros defined by the MAKEFLAGS environment variable, in the order
   specified in the environment variable. It is unspecified whether
   the internal macros defined in Internal Macros are accepted from
   this source.
3. The contents of the environment, excluding the MAKEFLAGS and SHELL
   variables and including the variables with null values.
4. Macros defined in the inference rules built into make.

Macro definitions from these sources shall not override macro defini?
tions from a lower-numbered source. Macro definitions from a single
source (for example, the make utility command line, the MAKEFLAGS envi?
ronment variable, or the other environment variables) shall override
previous macro definitions from the same source.

Macros defined in the makefile(s) shall override macro definitions that
occur before them in the makefile(s) and macro definitions from source
4. If the -e option is not specified, macros defined in the makefile(s)
shall override macro definitions from source 3. Macros defined in the
makefile(s) shall not override macro definitions from source 1 or
source 2.

Before the makefile(s) are read, all of the make utility command line
options (except -f and -p) and make utility command line macro defini?
tions (except any for the MAKEFLAGS macro), not already included in the
MAKEFLAGS macro, shall be added to the MAKEFLAGS macro, quoted in an
implementation-defined manner such that when MAKEFLAGS is read by an?

other instance of the make command, the original macro's value is re? covered. Other implementation-defined options and macros may also be added to the MAKEFLAGS macro. If this modifies the value of the MAKE? FLAGS macro, or, if the MAKEFLAGS macro is modified at any subsequent time, the MAKEFLAGS environment variable shall be modified to match the new value of the MAKEFLAGS macro. The result of setting MAKEFLAGS in the Makefile is unspecified.

Before the makefile(s) are read, all of the make utility command line macro definitions (except the MAKEFLAGS macro or the SHELL macro) shall be added to the environment of make. Other implementation-defined variables may also be added to the environment of make. Macros defined by the MAKEFLAGS environment variable and macros defined in the make? file(s) shall not be added to the environment of make if they are not already in its environment. With the exception of SHELL (see below), it is unspecified whether macros defined in these ways update the value of an environment variable that already exists in the environment of make.

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). 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 ef? fects of defining SHELL in the makefile or on the command line are im? plementation-defined.

Inference Rules

Inference rules are formatted as follows:

```
target:
<tab>command
[<tab>command]
...
line that does not begin with <tab> or #
```

The application shall ensure that the target portion is a valid target name (see Target Rules) 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 <slash> or <period> characters.) If there is only one <period> in the target, it is a sin? gle-suffix inference rule. Targets with two periods are double-suffix inference rules. Inference rules can have only one target before the <colon>.

The application shall ensure that the makefile does not specify prereq? uisites 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 in? ference 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 can also be formatted as follows:

```
rule: ;
```

where zero or more <blank> characters separate the <colon> and <semi? colon>.

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 shall 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 suf? fixes. 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 prereq? uisite with a suffix .s2 is specified as a target .s2.s1. The internal macros provide the means to specify general inference rules (see Inter? nal Macros).

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 com? pared to the list of suffixes specified by the .SUFFIXES special tar? gets. 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 re? spect 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 defini? tion of how to build target from target.s2. The other suffix (.s1) is treated as null.

A <tilde> ('~') in the above rules refers to an SCCS file in the cur? rent directory. Thus, the rule .c~.o would transform an SCCS C-lan? guage source file into an object file (.o). Because the s. of the SCCS files is a prefix, it is incompatible with make's suffix point of view. Hence, the '~' is a way of changing any file reference into an SCCS file reference.

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 application shall ensure that the member is an object file with the .o suffix. The modification time of the expression is the mod? ification time for the member as kept in the archive library; see ar.

The .a suffix shall refer to an archive library. The .s2.a rule shall

be used to update a member in the library from a file with a suffix .s2.

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-specified or make-specified rules for a particular class of target name. Explicit prerequisites are those prerequisites specified in a makefile on target lines. Implicit pre? requisites 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 accom? plished by recursively processing each prerequisite. Upon recursion, each prerequisite shall become a target itself. Its prerequisites in turn shall be processed recursively until a target is found that has no prerequisites, or further recursion would require applying two infer? ence rules one immediately after the other, at which point the recur? sion shall stop. As an extension, implementations may continue recur? sion when two or more successive inference rules need to be applied; however, if there are multiple different chains of such rules that could be used to create the target, it is unspecified which chain is used. The recursion shall then 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

* An implicit prerequisite that becomes a target when make processes it during recursion

In the definitions that follow, the word prerequisite refers to one of the following:

* An explicit prerequisite specified in the makefile for a particular target
* An implicit prerequisite generated as a result of locating an ap? propriate inference rule and corresponding file that matches the suffix of the target

The five internal macros are:

$@ The $@ shall evaluate to the full target name of the current target, or the archive filename part of a library archive tar? get. 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, the $< macro 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 eval? uate to the current target name. The meaning of the $< macro shall be otherwise unspecified.

For example, in the .c.a inference rule, $< represents the pre? requisite .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 alternative form. When an uppercase 'D' or 'F' is appended to any of the macros, the meaning shall be 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 shall be defined as:

$< member.s2

$* member

$@ lib

$? member.s2

$% member.o

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 FORTRAN may omit FC, FFLAGS, and the .f inference rules. Implementations may provide additional macros

and rules.

```
    SPECIAL TARGETS
    .SCCS_GET: sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@
    .SUFFIXES: .o .c .y .l .a .sh .f .c~ .y~ .l~ .sh~ .f~
    MACROS
    MAKE=make
    AR=ar
    ARFLAGS=-rv
    YACC=yacc
    YFLAGS=
    LEX=lex
    LFLAGS=
    LDFLAGS=
    CC=c99
    CFLAGS=-O 1
    FC=fort77
    FFLAGS=-O 1
    GET=get
    GFLAGS=
    SCCSFLAGS=
    SCCSGETFLAGS=-s
    SINGLE SUFFIX RULES
    .c:
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<
    .f:
        $(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<
    .sh:
        cp $< $@
        chmod a+x $@
    .c~:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c
    .f~:
```

```
	$(GET) $(GFLAGS) -p $< > $*.f
	$(FC) $(FFLAGS) $(LDFLAGS) -o $@ $*.f
.sh~:
	$(GET) $(GFLAGS) -p $< > $*.sh
	cp $*.sh $@
	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 $@
.c~.o:
	$(GET) $(GFLAGS) -p $< > $*.c
	$(CC) $(CFLAGS) -c $*.c
.f~.o:
	$(GET) $(GFLAGS) -p $< > $*.f
	$(FC) $(FFLAGS) -c $*.f
```

```
.y~.o:
    $(GET) $(GFLAGS) -p $< > $*.y
    $(YACC) $(YFLAGS) $*.y
    $(CC) $(CFLAGS) -c y.tab.c
    rm -f y.tab.c
    mv y.tab.o $@
.l~.o:
    $(GET) $(GFLAGS) -p $< > $*.l
    $(LEX) $(LFLAGS) $*.l
    $(CC) $(CFLAGS) -c lex.yy.c
    rm -f lex.yy.c
    mv lex.yy.o $@
.y~.c:
    $(GET) $(GFLAGS) -p $< > $*.y
    $(YACC) $(YFLAGS) $*.y
    mv y.tab.c $@
.l~.c:
    $(GET) $(GFLAGS) -p $< > $*.l
    $(LEX) $(LFLAGS) $*.l
    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
```

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.

CONSEQUENCES OF ERRORS

Default.

The following sections are informative.

APPLICATION USAGE

If there is a source file (such as ./source.c) and there are two SCCS files corresponding to it (./s.source.c and ./SCCS/s.source.c), on XSI-conformant systems make uses the SCCS file in the current directory. However, users are advised to use the underlying SCCS utilities (admin, delta, get, and so on) or the sccs utility for all source files in a given directory. If both forms are used for a given source file, future developers are very likely to be confused.

It is incumbent upon portable makefiles to specify the .POSIX special target in order to guarantee that they are not affected by local exten? sions.

The -k and -S options are both present so that the relationship between the command line, the MAKEFLAGS variable, and the makefile can be con? trolled 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.

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 vari? able, thus forbidding something like:

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

Many historical make utilities 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 best way to provide portable makefiles is to include all of the rules needed in the makefile itself. The rules provided use only fea? tures provided by other parts of this volume of POSIX.1?2017. The de? fault rules include rules for optional commands in this volume of POSIX.1?2017. Only rules pertaining to commands that are provided are needed in an implementation's default set.

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.

Some historical applications have been known to intermix target_name and macro=name operands on the command line, expecting that all of the macros are processed before any of the targets are dealt with. Conform? ing applications do not do this, although some backwards-compatibility support may be included in some implementations.

The following characters in filenames may give trouble: '=', ':', '`', single-quote, and '@'. In include filenames, pattern matching charac? ters and "" should also be avoided, as they may be treated as special by some implementations.

For inference rules, the description of $< and $? seem similar. How? ever, 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 is 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.

As a consequence of the general rules for target updating, a useful special case is that if a target has no prerequisites and no commands, and the target of the rule is a nonexistent file, then make acts as if this target has been updated whenever its rule is run.

Note: This implies that all targets depending on this one will al? ways have their commands run.

Shell command sequences like make; cp original copy; make may have problems on filesystems where the timestamp resolution is the minimum (1 second) required by the standard and where make considers identical timestamps to be up-to-date. Conversely, rules like copy: origi? nal; cp -p original copy will result in redundant work on make imple? mentations that consider identical timestamps to be out-of-date.

This standard does not specify precedence between macro definition and include directives. Thus, the behavior of:

```
include =foo.mk
```

is unspecified. To define a variable named include, either the white space before the <equal-sign> should be removed, or another macro should be used, as in:

```
INCLUDE_NAME = include
$(INCLUDE_NAME) =foo.mk
```

On the other hand, if the intent is to include a file which starts with an <equal-sign>, either the filename should be changed to ./=foo.mk, or the makefile should be written as:

```
INCLUDE_FILE = =foo.mk
include $(INCLUDE_FILE)
```

EXAMPLES

1. The following command:

   ```
   make
   ```

   makes the first target found in the makefile.

2. The following command:

make junk

makes the target junk.

3. 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:

```
.POSIX:
pgm: a.o b.o
    c99 a.o b.o -o pgm
a.o: incl.h a.c
    c99 -c a.c
b.o: incl.h b.c
    c99 -c b.c
```

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

```
.c.o:
    c99 -c -O 1 $*.c
```

or:

```
.c.o:
    c99 -c -O 1 $<
```

5. 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 in? sert them into lib.

The treatment of escaped <newline> characters throughout the make? file is historical practice. For example, the inference rule:

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

works, and the macro:

```
f= bar baz\
    biz
a:
    echo ==$f==
```

echoes "==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

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

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

RATIONALE

The make utility described in this volume of POSIX.1?2017 is intended to provide the means for changing portable source code into executables that can be run on an POSIX.1?2008-conforming 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 (such as from various multi-processor vendors, 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 the command lines of the target (BSD and others)
* Allowing <space> characters as well as <tab> characters 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 (BSD, System V, and most oth?
ers)
* Specifying an alternate shell to use to process commands.

Additionally, many vendors and research organizations have rethought the basic concepts of make, creating vastly extended, as well as com? pletely new, syntaxes. Each of these versions of make fulfills the needs of a different community of users; it is unreasonable for this volume of POSIX.1?2017 to require behavior that would be incompatible (and probably inferior) to historical practice for such a community.

In similar circumstances, when the industry has enough sufficiently in? compatible formats as to make them irreconcilable, this volume of POSIX.1?2017 has followed one or both of two courses of action. Com? mands have been renamed (cksum, echo, and pax) and/or command line op? tions 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 ``stan? dard'' behavior. This special target does not preclude extensions in the make utility, nor does it preclude such extensions being used by the makefile specifying the target; it does, however, preclude any ex? tensions from being applied that could alter the behavior of previously valid syntax; such extensions must be controlled via command line op? tions 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 reference page 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.

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 differ? ent to allow implementations to support both without ambiguity.

Early proposals 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 surround? ings.

The text about ``other implementation-defined pathnames may also be tried'' in addition to ./makefile and ./Makefile is to allow such ex? tensions as SCCS/s.Makefile and other variations. It was made an imple? mentation-defined requirement (as opposed to unspecified behavior) to highlight surprising implementations that might select something unex? pected like /etc/Makefile. XSI-conformant systems also try ./s.make? file, SCCS/s.makefile, ./s.Makefile, and SCCS/s.Makefile.

Early proposals 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 non-trivial extensions to makefile syntax. This ex? tension is not required by this volume of POSIX.1?2017, 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 im? plementors recognize the existing use of NPROC and/or PARALLEL as ex? tensions to make.

The default rules are based on System V. The default CC= value is c99 instead of cc because this volume of POSIX.1?2017 does not standardize the utility named cc. Thus, every conforming application would be re?

quired to define CC=c99 to expect to run. There is no advantage con? ferred by the hope that the makefile might hit the ``preferred'' com? piler because this cannot be guaranteed to work. Also, since the porta? ble makescript can only use the c99 options, no advantage is conferred in terms of what the script can do. It is a quality-of-implementation issue as to whether c99 is as valuable as cc.

The -d option to make is frequently used to produce debugging informa? tion, but is too implementation-defined to add to this volume of POSIX.1?2017.

The -p option is not passed in MAKEFLAGS on most historical implementa? tions 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 ap? ply to -q and -t as well. However, the System V convention of forcing command execution with -n when the command line of a target contains either of the strings "$(MAKE)" or "${MAKE}" has not been adopted. This functionality appeared in early proposals, but the danger of this ap? proach 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 situa? tion. 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 sys? tems 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 shall 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 pri? mary behavior.

One point of discussion was whether to drop the default rules list from this volume of POSIX.1?2017. They provide convenience, but do not en? hance 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, and related features provided by other make implementations, were omitted. In some implementations it is used to let 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. Further, a makefile writer cannot require an al? ternate shell to be used and still consider the makefile portable. While it would be possible to standardize a mechanism for specifying an alternate shell, existing implementations do not agree on such a mecha? nism, and makefile writers can already invoke an alternate shell by specifying the shell name in the rule for a target; for example:

```
python -c "foo"
```

The make utilities in most historical implementations process the pre? requisites of a target in left-to-right order, and the 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. Implementa? tions 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 makes allow comment lines (including blank and empty lines) within the collection of commands marked by leading <tab> characters. A conforming makefile must ensure that each command starts with a <tab>, but implementations are free to ignore comments without triggering the start of a new entry.

The ASYNCHRONOUS EVENTS section includes having SIGTERM and SIGHUP, along with the more traditional SIGINT and SIGQUIT, remove the current target unless directed not to do so. SIGTERM and SIGHUP were added to parallel other utilities that have historically cleaned up their work as a result of these signals. When make receives any signal other than SIGQUIT, it is required to resend itself the signal it received so that it exits 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 was to prevent damaged files from appearing up-to-date when make is rerun.

The .PRECIOUS special target was extended to affect all targets glob? ally (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 make in System V 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 conse? quences. The environment is the same as the environment to make except that MAKEFLAGS and macros defined on the make command line are added, and except that macros defined by the MAKEFLAGS environment variable and macros defined in the makefile(s) may update the value of an exist? ing environment variable (other than SHELL).

Some implementations do not use system() for all command lines, as re? quired by the 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 typi? cally 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 com? mand lines for a target and execute them using a single shell invoca? tion; the System V method is to pass each line individually to a sepa? rate shell. The single-shell method has the advantages in performance and the lack of a requirement for many continued lines. However, con? verting to this newer method has caused portability problems with many historical makefiles, so the behavior with the POSIX makefile is speci? fied to be the same as that of System V. It is suggested that the spe? cial 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>. Since it is often difficult to discern differences between <tab> and <space> characters on terminals or printed listings, confusing bugs can arise. In early proposals, an at? tempt was made to correct this problem by allowing leading <blank> characters instead of <tab> characters. 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 prac? tice of allowing leading <space> characters before macro definitions. Forcing macro lines into column 1 would be a significant backwards-com? patibility problem for some makefiles. Therefore, historical practice was restored.

There is substantial variation in the handling of include lines by dif? ferent implementations. However, there is enough commonality for the standard to be able to specify a minimum set of requirements that allow the feature to be used portably. Known variations have been explicitly called out as unspecified behavior in the description.

The System V dynamic dependency feature was not included. It would sup?

port:

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 up? date 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 avail? able from some vendors, but it would cause too many changes to this make to support. It would have introduced rule chaining and new substi? tution rules. However, the rules for target names have been set to re? serve the '%' and "" characters. These are traditionally used to im? plement metarules and quoting of target names, respectively. Implemen? tors are strongly encouraged to use these characters only for these purposes.

A request was made to extend the suffix delimiter character from a <pe? riod> to any character. The metarules feature in newer makes solves this problem in a more general way. This volume of POSIX.1?2017 is staying with the more conservative historical definition.

The standard output format for the -p option is not described because it is primarily a debugging option and because the format is not gener? ally 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=value operands on the command line, expecting that all of the macros are processed before any of the tar?

gets are dealt with. Conforming applications do not do this, but some backwards-compatibility support may be warranted.

Empty inference rules are specified with a <semicolon> command rather than omitting all commands, as described in an early proposal. The lat? ter case has no traditional meaning and is reserved for implementation extensions, such as in GNU make.

Earlier versions of this standard defined comment lines only as lines with '#' as the first character. Many places then talked about com? ments, blank lines, and empty lines; but some places inadvertently only mentioned comments when blank lines and empty lines had also been ac? cepted in all known implementations. The standard now defines comment lines to be blank lines, empty lines, and lines starting with a '#' character and explictily lists cases where blank lines and empty lines are not acceptable.

On most historic systems, the make utility considered a target with a prerequisite that had an identical timestamp as up-to-date. The HP-UX implementation of make treated it as out-of-date. The standard now al? lows either behavior, but implementations are encouraged to follow the example set by HP-UX. This is especially important on file systems where the timestamp resolution is the minimum (1 second) required by the standard. All implementations of make should make full use of the finest timestamp resolution available on the file systems holding tar? gets and prerequisites to ensure that targets are up-to-date even for prerequisite files with timestamps that were updated within the same second. However, if the timestamp resolutions of the file systems con? taining a target and a prerequisite are different, the timestamp with the more precise resolution should be rounded down to the resolution of the less precise timestamp for the comparison.

FUTURE DIRECTIONS

Some implementations of make include an export directive to add speci? fied make variables to the environment. This may be considered for standardization in a future version.

A future version of this standard may require that macro expansions us?

ing the forms $(string1:[op]%[os]=[np][%][ns]) or ${string1:[op]%[os]=[np][%][ns]} are treated as pattern macro expan? sions.

SEE ALSO

Chapter 2, Shell Command Language, ar, c99, get, lex, sccs, sh, yacc

The Base Definitions volume of POSIX.1?2017, Section 6.1, Portable Character Set, Chapter 8, Environment Variables, Section 12.2, Utility Syntax Guidelines

The System Interfaces volume of POSIX.1?2017, exec, system()

COPYRIGHT