NAME

make — maintain program dependencies

SYNOPSIS

make [**−BeikNnqrSstWwX**] [**−C** *directory*] [**−D** *variable*] [**−d** *flags*] [**−f** *makefile*] [**−I** *directory*] [**−J** *private*] [**−j** *max_jobs*] [**−m** *directory*] [**−T** *file*] [**−V** *variable*] [**−v** *variable*] [*variable* **=***value*] [*target* ...]

DESCRIPTION

make is a program designed to simplify the maintenance of other programs. Its input is a list of specifications as to the files upon which programs and other files depend. If no **−f** *makefile* option is given, **make** looks for the makefiles listed in *.MAKE.MAKEFILE_PREFERENCE* (default '*makefile*', '*Makefile*') in order to find the specifications. If the file '*.depend*' exists, it is read, see *mkdep*(1).

This manual page is intended as a reference document only. For a more thorough description of **make** and makefiles, please refer to *PMake − A Tutorial* (from 1993).

make prepends the contents of the *MAKEFLAGS* environment variable to the command line arguments before parsing them.

The options are as follows:

−B Try to be backwards compatible by executing a single shell per command and by making the sources of a dependency line in sequence.

−C *directory*
Change to *directory* before reading the makefiles or doing anything else. If multiple **−C** options are specified, each is interpreted relative to the previous one: **−C /** **−C** *etc* is equivalent to **−C** */etc*.

−D *variable*
Define *variable* to be 1, in the global scope.

−d [**−**]*flags*
Turn on debugging, and specify which portions of **make** are to print debugging information. Unless the flags are preceded by −, they are added to the *MAKEFLAGS* environment variable and are passed on to any child make processes. By default, debugging information is printed to standard error, but this can be changed using the **F** debugging flag. The debugging output is always unbuffered; in addition, if debugging is enabled but debugging output is not directed to standard output, the standard output is line buffered. The available *flags* are:

A Print all possible debugging information; equivalent to specifying all of the debugging flags.

a Print debugging information about archive searching and caching.

C Print debugging information about the current working directory.

c Print debugging information about conditional evaluation.

d Print debugging information about directory searching and caching.

e Print debugging information about failed commands and targets.

F[**+**]*filename*
Specify where debugging output is written. This must be the last flag, because it consumes the remainder of the argument. If the character immediately after the **F** flag is +, the file is opened in append mode; otherwise the file is overwritten. If the file name is `stdout` or `stderr`, debugging output is written to the standard output or standard error output respectively (and the + option has no effect). Otherwise, the output is written to the named file. If the file name ends with **.**`%d`, the `%d` is replaced by the pid.

f Print debugging information about loop evaluation.

g1 Print the input graph before making anything.

g2 Print the input graph after making everything, or before exiting on error.

g3 Print the input graph before exiting on error.

h Print debugging information about hash table operations.

j Print debugging information about running multiple shells.

L Turn on lint checks. This throws errors for variable assignments that do not parse correctly, at the time of assignment, so the file and line number are available.

l Print commands in Makefiles regardless of whether or not they are prefixed by `@` or other "quiet" flags. Also known as "loud" behavior.

M Print debugging information about "meta" mode decisions about targets.

m Print debugging information about making targets, including modification dates.

n Don't delete the temporary command scripts created when running commands. These temporary scripts are created in the directory referred to by the *TMPDIR* environment variable, or in */tmp* if *TMPDIR* is unset or set to the empty string. The temporary scripts are created by *mkstemp*(3), and have names of the form *makeXXXXXX*. *NOTE*: This can create many files in *TMPDIR* or */tmp*, so use with care.

p Print debugging information about makefile parsing.

s Print debugging information about suffix-transformation rules.

t Print debugging information about target list maintenance.

V Force the **−V** option to print raw values of variables, overriding the default behavior set via *.MAKE.EXPAND_VARIABLES*.

v Print debugging information about variable assignment and expansion.

x Run shell commands with **−x** so the actual commands are printed as they are executed.

−e Let environment variables override global variables within makefiles.

−f *makefile*

Specify a makefile to read instead of one of the defaults listed in *.MAKE.MAKEFILE_PREFERENCE*. If *makefile* is −, standard input is read. If *makefile* starts with the string **. . . /**, **make** searches for the specified path in the rest of the argument in the current directory and its parents. Multiple makefiles may be specified, and are read in the order specified.

−I *directory*

Specify a directory in which to search for makefiles and included makefiles. The system makefile directory (or directories, see the **−m** option) is automatically included as part of this list.

−i Ignore non-zero exit of shell commands in the makefile. Equivalent to specifying − before each command line in the makefile.

−J *private*

This option should *not* be specified by the user.

When the **−j** option is in use in a recursive build, this option is passed by a make to child makes to allow all the make processes in the build to cooperate to avoid overloading the system.

−j *max_jobs*

Specify the maximum number of jobs that **make** may have running at any one time. If *max_jobs* is a floating point number, or ends with `C`, then the value is multiplied by the number of CPUs reported online by *sysconf*(3). The value of *max_jobs* is saved in *.MAKE.JOBS*. Turns compatibility mode off, unless the **−B** option is also specified. When compatibility mode is off, all commands associated with a target are executed in a single shell invocation as opposed to the traditional one shell invocation per line. This can break traditional scripts which change directories on each command invocation and then expect to start with a fresh environment on the next line. It is more efficient to correct the scripts rather than turn backwards compatibility on.

A job token pool with *max_jobs* tokens is used to control the total number of jobs running. Each instance of **make** will wait for a token from the pool before running a new job.

−k Continue processing after errors are encountered, but only on those targets that do not depend on the target whose creation caused the error.

−m *directory*

Specify a directory in which to search for *sys.mk* and makefiles included via the **<***file***>**-style include statement. The **−m** option can be used multiple times to form a search path. This path overrides the default system include path */usr/share/mk*. Furthermore, the system include path is appended to the search path used for **"***file***"**-style include statements (see the **−I** option). The system include path can be referenced via the read-only variable *.SYSPATH*.

If a directory name in the **−m** argument (or the *MAKESYSPATH* environment variable) starts with the string `...`/, **make** searches for the specified file or directory named in the remaining part of the argument string. The search starts with the current directory and then works upward towards the root of the file system. If the search is successful, the resulting directory replaces the `...`/ specification in the **−m** argument. This feature allows **make** to easily search in the current source tree for customized *sys.mk* files (e.g., by using `...`/`mk`/`sys`.`mk` as an argument).

−n Display the commands that would have been executed, but do not actually execute them unless the target depends on the *.MAKE* special source (see below) or the command is prefixed with '**+**'.

−N Display the commands that would have been executed, but do not actually execute any of them; useful for debugging top-level makefiles without descending into subdirectories.

−q Do not execute any commands, instead exit 0 if the specified targets are up to date, and 1 otherwise.

−r Do not use the built-in rules specified in the system makefile.

−S Stop processing if an error is encountered. This is the default behavior and the opposite of **−k**.

−s Do not echo any commands as they are executed. Equivalent to specifying '@' before each command line in the makefile.

−T *tracefile*

When used with the **−j** flag, append a trace record to *tracefile* for each job started and completed.

−t Rather than re-building a target as specified in the makefile, create it or update its modification time to make it appear up-to-date.

−V *variable*

Print the value of *variable*. Do not build any targets. Multiple instances of this option may be specified; the variables are printed one per line, with a blank line for each null or undefined variable. The value printed is extracted from the global scope after all makefiles have been read.

By default, the raw variable contents (which may include additional unexpanded variable references) are shown. If *variable* contains a `$`, it is not interpreted as a variable name but rather as an expression. Its value is expanded before printing. The value is also expanded before printing if *.MAKE.EXPAND_VARIABLES* is set to true and the **−dV** option has not been used to override it.

Note that loop-local and target-local variables, as well as values taken temporarily by global variables during makefile processing, are not accessible via this option. The **−dv** debug mode can be used to see these at the cost of generating substantial extraneous output.

−v *variable*
Like **−V**, but all printed variables are always expanded to their complete value. The last occurrence of **−V** or **−v** decides whether all variables are expanded or not.

−W Treat any warnings during makefile parsing as errors.

−w Print entering and leaving directory messages, pre and post processing.

−X Don't export variables passed on the command line to the environment individually. Variables passed on the command line are still exported via the *MAKEFLAGS* environment variable. This option may be useful on systems which have a small limit on the size of command arguments.

*variable***=***value*
Set the value of the variable *variable* to *value*. Normally, all values passed on the command line are also exported to sub-makes in the environment. The **−X** flag disables this behavior. Variable assignments should follow options for POSIX compatibility but no ordering is enforced.

There are several different types of lines in a makefile: dependency specifications, shell commands, variable assignments, include statements, conditional directives, for loops, other directives, and comments.

Lines may be continued from one line to the next by ending them with a backslash (\). The trailing newline character and initial whitespace on the following line are compressed into a single space.

FILE DEPENDENCY SPECIFICATIONS

Dependency lines consist of one or more targets, an operator, and zero or more sources. This creates a relationship where the targets "depend" on the sources and are customarily created from them. A target is considered out of date if it does not exist, or if its modification time is less than that of any of its sources. An out-of-date target is re-created, but not until all sources have been examined and themselves re-created as needed. Three operators may be used:

: Many dependency lines may name this target but only one may have attached shell commands. All sources named in all dependency lines are considered together, and if needed the attached shell commands are run to create or re-create the target. If **make** is interrupted, the target is removed.

! The same, but the target is always re-created whether or not it is out of date.

:: Any dependency line may have attached shell commands, but each one is handled independently: its sources are considered and the attached shell commands are run if the target is out of date with respect to (only) those sources. Thus, different groups of the attached shell commands may be run depending on the circumstances. Furthermore, unlike **:**, for dependency lines with no sources, the attached shell commands are always run. Also unlike **:**, the target is not removed if **make** is interrupted.

All dependency lines mentioning a particular target must use the same operator.

Targets and sources may contain the shell wildcard values `?`, `*`, `[]`, and `{}`. The values `?`, `*`, and `[]` may only be used as part of the final component of the target or source, and only match existing files. The value `{}` need not necessarily be used to describe existing files. Expansion is in directory order, not alphabetically as done in the shell.

SHELL COMMANDS

Each target may have associated with it one or more lines of shell commands, normally used to create the target. Each of the lines in this script *must* be preceded by a tab. (For historical reasons, spaces are not accepted.) While targets can occur in many dependency lines if desired, by default only one of these rules may be followed by a creation script. If the '**::**' operator is used, however, all rules may include scripts, and the respective scripts are executed in the order found.

Each line is treated as a separate shell command, unless the end of line is escaped with a backslash \, in which case that line and the next are combined. If the first characters of the command are any combination of '**@**', '**+**', or '**−**', the command is treated specially.

@ causes the command not to be echoed before it is executed.

+ causes the command to be executed even when **−n** is given. This is similar to the effect of the *.MAKE* special source, except that the effect can be limited to a single line of a script.

− in compatibility mode causes any non-zero exit status of the command line to be ignored.

When **make** is run in jobs mode with **−j** *max_jobs*, the entire script for the target is fed to a single instance of the shell. In compatibility (non-jobs) mode, each command is run in a separate process. If the command contains any shell meta characters (`#=|^(){};&<>*?[]:$'\\n`), it is passed to the shell; otherwise **make** attempts direct execution. If a line starts with '**−**' and the shell has ErrCtl enabled, failure of the command line is ignored as in compatibility mode. Otherwise '**−**' affects the entire job; the script stops at the first command line that fails, but the target is not deemed to have failed.

Makefiles should be written so that the mode of **make** operation does not change their behavior. For example, any command which uses "cd" or "chdir" without the intention of changing the directory for subsequent commands should be put in parentheses so it executes in a subshell. To force the use of a single shell, escape the line breaks so as to make the whole script one command. For example:

```
avoid-chdir-side-effects:
        @echo "Building $@ in $$(pwd)"
        @(cd ${.CURDIR} && ${MAKE} $@)
        @echo "Back in $$(pwd)"

ensure-one-shell-regardless-of-mode:
        @echo "Building $@ in $$(pwd)"; \
        (cd ${.CURDIR} && ${MAKE} $@); \
        echo "Back in $$(pwd)"
```

Since **make** changes the current working directory to '*.OBJDIR*' before executing any targets, each child process starts with that as its current working directory.

VARIABLE ASSIGNMENTS

Variables in make behave much like macros in the C preprocessor.

Variable assignments have the form '*NAME op value*', where:

NAME is a single-word variable name, consisting, by tradition, of all upper-case letters,

op is one of the variable assignment operators described below, and

value is interpreted according to the variable assignment operator.

Whitespace around *NAME*, *op* and *value* is discarded.

Variable assignment operators

The five operators that assign values to variables are:

= Assign the value to the variable. Any previous value is overwritten.

+= Append the value to the current value of the variable, separating them by a single space.

?= Assign the value to the variable if it is not already defined.

:= Expand the value, then assign it to the variable.

NOTE: References to undefined variables are *not* expanded. This can cause problems when variable modifiers are used.

!= Expand the value and pass it to the shell for execution, then assign the output from the child's standard output to the variable. Any newlines in the result are replaced with spaces.

Expansion of variables

In most contexts where variables are expanded, `$$` expands to a single dollar sign. In other contexts (most variable modifiers, string literals in conditions), `\$` expands to a single dollar sign.

References to variables have the form **${***name*[**:***modifiers*]**}** or **$(***name*[**:***modifiers*]**)**. If the variable name consists of only a single character and the expression contains no modifiers, the surrounding curly braces or parentheses are not required. This shorter form is not recommended.

If the variable name contains a dollar, the name itself is expanded first. This allows almost arbitrary variable names, however names containing dollar, braces, parentheses or whitespace are really best avoided.

If the result of expanding a nested variable expression contains a dollar sign (`$`), the result is subject to further expansion.

Variable substitution occurs at four distinct times, depending on where the variable is being used.

1. Variables in dependency lines are expanded as the line is read.
2. Variables in conditionals are expanded individually, but only as far as necessary to determine the result of the conditional.
3. Variables in shell commands are expanded when the shell command is executed.
4. **.for** loop index variables are expanded on each loop iteration. Note that other variables are not expanded when composing the body of a loop, so the following example code:

```
.for i in 1 2 3
a+=     ${i}
j=      ${i}
b+=     ${j}
.endfor

all:
        @echo ${a}
        @echo ${b}
```

prints:

```
1 2 3
3 3 3
```

After the loop is executed:

a contains `${:U1} ${:U2} ${:U3}`, which expands to `1 2 3`.

j contains `${:U3}`, which expands to `3`.

b contains `${j} ${j} ${j}`, which expands to `${:U3} ${:U3} ${:U3}` and further to `3 3 3`.

Variable classes

The four different classes of variables (in order of increasing precedence) are:

Environment variables
: Variables defined as part of **make**'s environment.

Global variables
: Variables defined in the makefile or in included makefiles.

Command line variables
: Variables defined as part of the command line.

Local variables
: Variables that are defined specific to a certain target.

Local variables can be set on a dependency line, unless *.MAKE.TARGET_LOCAL_VARIABLES* is set to `false`. The rest of the line (which already has had global variables expanded) is the variable value. For example:

```
COMPILER_WRAPPERS= ccache distcc icecc

${OBJS}: .MAKE.META.CMP_FILTER=${COMPILER_WRAPPERS:S,^,N,}
```

Only the targets `${OBJS}` are impacted by that filter (in "meta" mode) and simply enabling/disabling any of the compiler wrappers does not render all of those targets out-of-date.

NOTE: target-local variable assignments behave differently in that;

+= Only appends to a previous local assignment for the same target and variable.

:= Is redundant with respect to global variables, which have already been expanded.

The built-in local variables are:

.ALLSRC The list of all sources for this target; also known as '*>*' or '*^*'.

.ARCHIVE The name of the archive file; also known as '*!*'.

.IMPSRC In suffix-transformation rules, the name/path of the source from which the target is to be transformed (the "implied" source); also known as '*<*'. It is not defined in explicit rules.

.MEMBER The name of the archive member; also known as '*%*'.

.OODATE The list of sources for this target that were deemed out-of-date; also known as '*?*'.

.PREFIX The name of the target with suffix (if declared in **.SUFFIXES**) removed; also known as '***'.

.TARGET The name of the target; also known as '*@*'. For compatibility with other makes this is an alias for *.ARCHIVE* in archive member rules.

The shorter forms ('*>*', '*^*', '*!*', '*<*', '*%*', '*?*', '***', and '*@*') are permitted for backward compatibility with historical makefiles and legacy POSIX make and are not recommended.

Variants of these variables with the punctuation followed immediately by `D` or `F`, e.g. `$(@D)`, are legacy forms equivalent to using the `:H` and `:T` modifiers. These forms are accepted for compatibility with AT&T System V UNIX makefiles and POSIX but are not recommended.

Four of the local variables may be used in sources on dependency lines because they expand to the proper value for each target on the line. These variables are '*.TARGET*', '*.PREFIX*', '*.ARCHIVE*', and '*.MEMBER*'.

Additional built-in variables

In addition, **make** sets or knows about the following variables:

.ALLTARGETS
: The list of all targets encountered in the makefiles. If evaluated during makefile parsing, lists only those targets encountered thus far.

.CURDIR
: A path to the directory where **make** was executed. Refer to the description of '*PWD*' for more details.

.ERROR_CMD
Is used in error handling, see *MAKE_PRINT_VAR_ON_ERROR*.

.ERROR_CWD
Is used in error handling, see *MAKE_PRINT_VAR_ON_ERROR*.

.ERROR_EXIT
Is used in error handling, see *MAKE_PRINT_VAR_ON_ERROR*.

.ERROR_META_FILE
Is used in error handling in "meta" mode, see *MAKE_PRINT_VAR_ON_ERROR*.

.ERROR_TARGET
Is used in error handling, see *MAKE_PRINT_VAR_ON_ERROR*.

.INCLUDEDFROMDIR
The directory of the file this makefile was included from.

.INCLUDEDFROMFILE
The filename of the file this makefile was included from.

MACHINE
The machine hardware name, see *uname*(1).

MACHINE_ARCH
The machine processor architecture name, see *uname*(1).

MAKE The name that **make** was executed with (*argv[0]*).

.MAKE The same as *MAKE*, for compatibility. The preferred variable to use is the environment variable *MAKE* because it is more compatible with other make variants and cannot be confused with the special target with the same name.

.MAKE.ALWAYS_PASS_JOB_QUEUE
Tells **make** whether to pass the descriptors of the job token queue even if the target is not tagged with **.MAKE** The default is `yes` for backwards compatability with FreeBSD 9.0 and earlier.

.MAKE.DEPENDFILE
Names the makefile (default '*.depend*') from which generated dependencies are read.

.MAKE.DIE_QUIETLY
If set to `true`, do not print error information at the end.

.MAKE.EXPAND_VARIABLES
A boolean that controls the default behavior of the **−V** option. If true, variable values printed with **−V** are fully expanded; if false, the raw variable contents (which may include additional unexpanded variable references) are shown.

.MAKE.EXPORTED
The list of variables exported by **make**.

MAKE_VERSION
This variable indicates the version of **make**. It is typically the date of last import from NetBSD. It is useful for checking whether certain features are available.

MAKEFILE
The top-level makefile that is currently read, as given in the command line.

.MAKEFLAGS
The environment variable '*MAKEFLAGS*' may contain anything that may be specified on **make**'s command line. Anything specified on **make**'s command line is appended to the *.MAKEFLAGS* variable, which is then added to the environment for all programs that **make** executes.

.MAKE.GID
The numeric group ID of the user running **make**. It is read-only.

.MAKE.JOB.PREFIX
If **make** is run with **–j**, the output for each target is prefixed with a token
`--- target ---`
the first part of which can be controlled via *.MAKE.JOB.PREFIX*. If *.MAKE.JOB.PREFIX* is empty, no token is printed. For example, setting *.MAKE.JOB.PREFIX* to `${.newline}---${.MAKE:T}[${.MAKE.PID}]` would produce tokens like
`---make[1234] target ---`
making it easier to track the degree of parallelism being achieved.

.MAKE.JOBS
The argument to the **–j** option.

.MAKE.JOBS.C
A read-only boolean that indicates whether the **–j** option supports use of `C`.

.MAKE.LEVEL
The recursion depth of **make**. The top-level instance of **make** has level 0, and each child make has its parent level plus 1. This allows tests like: **.if ${.MAKE.LEVEL} == 0** to protect things which should only be evaluated in the top-level instance of **make**.

.MAKE.LEVEL.ENV
The name of the environment variable that stores the level of nested calls to **make**.

.MAKE.MAKEFILE_PREFERENCE
The ordered list of makefile names (default '*makefile*', '*Makefile*') that **make** looks for.

.MAKE.MAKEFILES
The list of makefiles read by **make**, which is useful for tracking dependencies. Each makefile is recorded only once, regardless of the number of times read.

.MAKE.META.BAILIWICK
In "meta" mode, provides a list of prefixes which match the directories controlled by **make**. If a file that was generated outside of *.OBJDIR* but within said bailiwick is missing, the current target is considered out-of-date.

.MAKE.META.CMP_FILTER
In "meta" mode, it can (very rarely!) be useful to filter command lines before comparison. This variable can be set to a set of modifiers that are applied to each line of the old and new command that differ, if the filtered commands still differ, the target is considered out-of-date.

.MAKE.META.CREATED
In "meta" mode, this variable contains a list of all the meta files updated. If not empty, it can be used to trigger processing of *.MAKE.META.FILES*.

.MAKE.META.FILES
In "meta" mode, this variable contains a list of all the meta files used (updated or not). This list can be used to process the meta files to extract dependency information.

.MAKE.META.IGNORE_FILTER
Provides a list of variable modifiers to apply to each pathname. Ignore if the expansion is an empty string.

.MAKE.META.IGNORE_PATHS
Provides a list of path prefixes that should be ignored; because the contents are expected to change over time. The default list includes: '*/dev /etc /proc /tmp /var/run /var/tmp*'

.MAKE.META.IGNORE_PATTERNS
Provides a list of patterns to match against pathnames. Ignore any that match.

.MAKE.META.PREFIX
Defines the message printed for each meta file updated in "meta verbose" mode. The default value is:

```
Building ${.TARGET:H:tA}/${.TARGET:T}
```

.MAKE.MODE
Processed after reading all makefiles. Affects the mode that **make** runs in. It can contain these keywords:

compat Like **−B**, puts **make** into "compat" mode.

meta Puts **make** into "meta" mode, where meta files are created for each target to capture the commands run, the output generated, and if *filemon*(4) is available, the system calls which are of interest to **make**. The captured output can be useful when diagnosing errors.

make will use the information in the meta file to help determine if a target is out-of-date when the normal dependency rules indicate it is not.

First, the commands to be executed, will be compared to those captured previously, if any differ, the target is out-of-date.

This allows for a huge improvement in the reliability and efficiency of update builds. It is no longer necessary for targets to depend on makefiles just in-case they set a variable that might be relevant. Mechanisms such as *.MAKE.META.CMP_FILTER* and **.NOMETA_CMP**, allow limiting or disabling that comparison on a per target basis. A reference to the variable *.OODATE* can be leveraged to block comparison of certain commands. For example: `${.OODATE:M}` will expand to nothing and have no impact on the target, its side-effect though, will be to prevent comparison of any command line it appears on. For documentation purposes `${.OODATE:MNOMETA_CMP}` is useful.

If necessary, **make** will then use the information captured by *filemon*(4), to check the modification time of any file used in generating the target, if any is newer, the target is out-of-date.

Such deep inspection can easily lead to cases where a target is *always* considered out-of-date, which is why *.MAKE.META.IGNORE_FILTER*, *.MAKE.META.IGNORE_PATHS* and *.MAKE.META.IGNORE_PATTERNS*, are provided to limit that inspection when necessary.

curdirOk=*bf*
By default, **make** does not create *.meta* files in '*.CURDIR*'. This can be overridden by setting *bf* to a value which represents true.

missing-meta=*bf*
If *bf* is true, a missing *.meta* file makes the target out-of-date.

missing-filemon=*bf*
If *bf* is true, missing filemon data makes the target out-of-date.

nofilemon
Do not use *filemon*(4).

env For debugging, it can be useful to include the environment in the *.meta* file.

verbose If in "meta" mode, print a clue about the target being built. This is useful if the build is otherwise running silently. The message printed is the expanded value of *.MAKE.META.PREFIX*.

ignore-cmd
Some makefiles have commands which are simply not stable. This keyword causes them to be ignored for determining whether a target is out of date in "meta" mode. See also **.NOMETA_CMP**.

silent=*bf*
If *bf* is true, when a .meta file is created, mark the target **.SILENT**.

randomize-targets
In both compat and parallel mode, do not make the targets in the usual order, but instead randomize their order. This mode can be used to detect undeclared dependencies between files.

MAKEOBJDIR
Used to create files in a separate directory, see *.OBJDIR*.

MAKE_OBJDIR_CHECK_WRITABLE
When true, **make** will check that *.OBJDIR* is writable, and issue a warning if not.

MAKE_DEBUG_OBJDIR_CHECK_WRITABLE
When true and **make** is warning about an unwritable *.OBJDIR*, report the variables listed in *MAKE_PRINT_VAR_ON_ERROR* to help debug.

MAKEOBJDIRPREFIX
Used to create files in a separate directory, see *.OBJDIR*. It should be an absolute path.

.MAKE.OS
The name of the operating system, see *uname*(1). It is read-only.

.MAKEOVERRIDES
This variable is used to record the names of variables assigned to on the command line, so that they may be exported as part of '*MAKEFLAGS*'. This behavior can be disabled by assigning an empty value to '*.MAKEOVERRIDES*' within a makefile. Extra variables can be exported from a makefile by appending their names to '*.MAKEOVERRIDES*'. '*MAKEFLAGS*' is re-exported whenever '*.MAKEOVERRIDES*' is modified.

.MAKE.PATH_FILEMON
If **make** was built with *filemon*(4) support, this is set to the path of the device node. This allows makefiles to test for this support.

.MAKE.PID
The process ID of **make**. It is read-only.

.MAKE.PPID
The parent process ID of **make**. It is read-only.

MAKE_PRINT_VAR_ON_ERROR
When **make** stops due to an error, it sets '*.ERROR_TARGET*' to the name of the target that failed, '*.ERROR_EXIT*' to the exit status of the failed target, '*.ERROR_CMD*' to the commands of the failed target, and in "meta" mode, it also sets '*.ERROR_CWD*' to the *getcwd*(3), and '*.ERROR_META_FILE*' to the path of the meta file (if any) describing the failed target. It then prints its name and the value of '*.CURDIR*' as well as the value of any variables named in '*MAKE_PRINT_VAR_ON_ERROR*'.

.MAKE.SAVE_DOLLARS
If true, `$$` are preserved when doing `:=` assignments. The default is false, for backwards compatibility. Set to true for compatability with other makes. If set to false, `$$` becomes `$` per normal evaluation rules.

.MAKE.TARGET_LOCAL_VARIABLES
If set to `false`, apparent variable assignments in dependency lines are treated as normal sources.

.MAKE.UID
The numeric ID of the user running **make**. It is read-only.

.newline
This variable is simply assigned a newline character as its value. It is read-only. This allows expansions using the **:@** modifier to put a newline between iterations of the loop rather than a space. For example, in case of an error, **make** prints the variable names and their values using:

```
${MAKE_PRINT_VAR_ON_ERROR:@v@$v='${$v}'${.newline}@}
```

.OBJDIR
A path to the directory where the targets are built. Its value is determined by trying to *chdir*(2) to the following directories in order and using the first match:

1. **${MAKEOBJDIRPREFIX}${.CURDIR}**

   (Only if '*MAKEOBJDIRPREFIX*' is set in the environment or on the command line.)

2. **${MAKEOBJDIR}**

   (Only if '*MAKEOBJDIR*' is set in the environment or on the command line.)

3. **${.CURDIR}***/obj.***${MACHINE}**

4. **${.CURDIR}***/obj*

5. */usr/obj/***${.CURDIR}**

6. **${.CURDIR}**

Variable expansion is performed on the value before it is used, so expressions such as **${.CURDIR:S,^/usr/src,/var/obj,}** may be used. This is especially useful with '*MAKEOBJDIR*'.

'*.OBJDIR*' may be modified in the makefile via the special target '**.OBJDIR**'. In all cases, **make** changes to the specified directory if it exists, and sets '*.OBJDIR*' and '*PWD*' to that directory before executing any targets.

Except in the case of an explicit '**.OBJDIR**' target, **make** checks that the specified directory is writable and ignores it if not. This check can be skipped by setting the environment variable '*MAKE_OBJDIR_CHECK_WRITABLE*' to "no".

.PARSEDIR
The directory name of the current makefile being parsed.

.PARSEFILE
The basename of the current makefile being parsed. This variable and '*.PARSEDIR*' are both set only while the makefiles are being parsed. To retain their current values, assign them to a variable using assignment with expansion '**:=**'.

.PATH The space-separated list of directories that **make** searches for files. To update this search list, use the special target '**.PATH**' rather than modifying the variable directly.

%POSIX
Is set in POSIX mode, see the special `.POSIX` target.

PWD Alternate path to the current directory. **make** normally sets '*.CURDIR*' to the canonical path given by *getcwd*(3). However, if the environment variable '*PWD*' is set and gives a path to the current directory, **make** sets '*.CURDIR*' to the value of '*PWD*' instead. This behavior is disabled if '*MAKEOBJDIRPREFIX*' is set or '*MAKEOBJDIR*' contains a variable transform. '*PWD*' is set to the value of '*.OBJDIR*' for all programs which **make** executes.

.SHELL
The pathname of the shell used to run target scripts. It is read-only.

.SUFFIXES
The list of known suffixes. It is read-only.

.SYSPATH
The space-separated list of directories that **make** searches for makefiles, referred to as the system include path. To update this search list, use the special target '**.SYSPATH**' rather than modifying the variable which is read-only.

.TARGETS
The list of targets explicitly specified on the command line, if any.

VPATH The colon-separated (":") list of directories that **make** searches for files. This variable is supported for compatibility with old make programs only, use '*.PATH*' instead.

Variable modifiers

The general format of a variable expansion is:

${*variable* [**:***modifier* [**:** . . .]]**}**

Each modifier begins with a colon. To escape a colon, precede it with a backslash `\`.

A list of indirect modifiers can be specified via a variable, as follows:

```
modifier_variable = modifier[:...]

${variable:${modifier_variable}[:...]}
```

In this case, the first modifier in the *modifier_variable* does not start with a colon, since that colon already occurs in the referencing variable. If any of the modifiers in the *modifier_variable* contains a dollar sign (`$`), these must be doubled to avoid early expansion.

Some modifiers interpret the expression value as a single string, others treat the expression value as a white-space-separated list of words. When splitting a string into words, whitespace can be escaped using double quotes, single quotes and backslashes, like in the shell. The quotes and backslashes are retained in the words.

The supported modifiers are:

:E Replaces each word with its suffix.

:H Replaces each word with its dirname.

:M*pattern*
Selects only those words that match *pattern*. The standard shell wildcard characters (`*`, `?`, and `[]`) may be used. The wildcard characters may be escaped with a backslash (`\`). As a consequence of the way values are split into words, matched, and then joined, the construct `${VAR:M*}` removes all leading and trailing whitespace and normalizes the inter-word spacing to a single space.

:N*pattern*
This is the opposite of '**:M**', selecting all words which do *not* match *pattern*.

:O Orders the words lexicographically.

:On Orders the words numerically. A number followed by one of `k`, `M` or `G` is multiplied by the appropriate factor, which is 1024 for `k`, 1048576 for `M`, or 1073741824 for `G`. Both upper- and lower-case letters are accepted.

:Or Orders the words in reverse lexicographical order.

:Orn
Orders the words in reverse numerical order.

:Ox Shuffles the words. The results are different each time you are referring to the modified variable; use the assignment with expansion '**:=**' to prevent such behavior. For example,

```
LIST=                   uno due tre quattro
RANDOM_LIST=            ${LIST:Ox}
STATIC_RANDOM_LIST:=    ${LIST:Ox}

all:
        @echo "${RANDOM_LIST}"
        @echo "${RANDOM_LIST}"
        @echo "${STATIC_RANDOM_LIST}"
        @echo "${STATIC_RANDOM_LIST}"
```

may produce output similar to:

```
quattro due tre uno
tre due quattro uno
due uno quattro tre
due uno quattro tre
```

:Q Quotes every shell meta-character in the value, so that it can be passed safely to the shell.

:q Quotes every shell meta-character in the value, and also doubles '$' characters so that it can be passed safely through recursive invocations of **make**. This is equivalent to '**:S/\$/&&/g:Q**'.

:R Replaces each word with everything but its suffix.

:range[**=***count*]
The value is an integer sequence representing the words of the original value, or the supplied *count*.

:gmtime[**=***timestamp*]
The value is interpreted as a format string for *strftime*(3), using *gmtime*(3), producing the formatted timestamp. Note: the `%s` format should only be used with '**:localtime**'. If a *timestamp* value is not provided or is 0, the current time is used.

:hash
Computes a 32-bit hash of the value and encodes it as 8 hex digits.

:localtime[**=***timestamp*]
The value is interpreted as a format string for *strftime*(3), using *localtime*(3), producing the formatted timestamp. If a *timestamp* value is not provided or is 0, the current time is used.

:mtime[**=***timestamp*]
Call *stat*(2) with each word as pathname; use `st_mtime` as the new value. If *stat*(2) fails; use *timestamp* or current time. If *timestamp* is set to `error`, then *stat*(2) failure will cause an error.

:tA Attempts to convert the value to an absolute path using *realpath*(3). If that fails, the value is unchanged.

:tl Converts the value to lower-case letters.

:ts*c* When joining the words after a modifier that treats the value as words, the words are normally separated by a space. This modifier changes the separator to the character *c*. If *c* is omitted, no separator is used. The common escapes (including octal numeric codes) work as expected.

:tt Converts the first character of each word to upper-case, and the rest to lower-case letters.

:tu Converts the value to upper-case letters.

:tW Causes subsequent modifiers to treat the value as a single word (possibly containing embedded whitespace). See also '**:[***]'.

:tw Causes the value to be treated as a list of words. See also '**:[@]**'.

:S /*old_string* /*new_string* /[**1gW**]
Modifies the first occurrence of *old_string* in each word of the value, replacing it with *new_string*. If a `g` is appended to the last delimiter of the pattern, all occurrences in each word are replaced. If a `1` is appended to the last delimiter of the pattern, only the first occurrence is affected. If a `W` is appended to the last delimiter of the pattern, the value is treated as a single word. If *old_string* begins with a caret (`^`), *old_string* is anchored at the beginning of each word. If *old_string* ends with a dollar sign (`$`), it is anchored at the end of each word. Inside *new_string*, an ampersand (`&`) is replaced by *old_string* (without the anchoring `^` or `$`). Any character may be used as the delimiter for the parts of the modifier string. The anchoring, ampersand and delimiter characters can be escaped with a backslash (`\`).

Both *old_string* and *new_string* may contain nested expressions. To prevent a dollar sign from starting a nested expression, escape it with a backslash.

:C /*pattern* /*replacement* /[**1gW**]
The **:C** modifier works like the **:S** modifier except that the old and new strings, instead of being simple strings, are an extended regular expression *pattern* (see *regex*(3)) and an *ed*(1)−style *replacement*. Normally, the first occurrence of the pattern *pattern* in each word of the value is substituted with *replacement*. The `1` modifier causes the substitution to apply to at most one word; the `g` modifier causes the substitution to apply to as many instances of the search pattern *pattern* as occur in the word or words it is found in; the `W` modifier causes the value to be treated as a single word (possibly containing embedded whitespace).

As for the **:S** modifier, the *pattern* and *replacement* are subjected to variable expansion before being parsed as regular expressions.

:T Replaces each word with its last path component (basename).

:u Removes adjacent duplicate words (like *uniq*(1)).

:? *true_string* **:***false_string*
If the variable name (not its value), when parsed as a **.if** conditional expression, evaluates to true, return as its value the *true_string*, otherwise return the *false_string*. Since the variable name is used as the expression, :? must be the first modifier after the variable name itself—which, of course, usually contains variable expansions. A common error is trying to use expressions like

```
${NUMBERS:M42:?match:no}
```

which actually tests defined(NUMBERS). To determine if any words match "42", you need to use something like:

```
${"${NUMBERS:M42}" != "":?match:no}.
```

: *old_string* **=***new_string*
This is the AT&T System V UNIX style substitution. It can only be the last modifier specified, as a **:** in either *old_string* or *new_string* is treated as a regular character, not as the end of the modifier.

If *old_string* does not contain the pattern matching character `%`, and the word ends with *old_string* or equals it, that suffix is replaced with *new_string*.

Otherwise, the first `%` in *old_string* matches a possibly empty substring of arbitrary characters, and if the whole pattern is found in the word, the matching part is replaced with *new_string*, and the first occurrence of `%` in *new_string* (if any) is replaced with the substring matched by the `%`.

Both *old_string* and *new_string* may contain nested expressions. To prevent a dollar sign from starting a nested expression, escape it with a backslash.

:@*varname* **@***string* **@**

This is the loop expansion mechanism from the OSF Development Environment (ODE) make. Unlike **.for** loops, expansion occurs at the time of reference. For each word in the value, assign the word to the variable named *varname* and evaluate *string*. The ODE convention is that *varname* should start and end with a period, for example:

```
${LINKS:@.LINK.@${LN} ${TARGET} ${.LINK.}@}
```

However, a single-letter variable is often more readable:

```
${MAKE_PRINT_VAR_ON_ERROR:@v@$v='${$v}'${.newline}@}
```

:_[**=***var*]

Saves the current variable value in `$_` or the named *var* for later reference. Example usage:

```
M_cmpv.units = 1 1000 1000000
M_cmpv = S,., ,g:_:range:@i@+ $${_:[-$$i]} \
\* $${M_cmpv.units:[$$i]}@:S,^,expr 0 ,1:sh

.if ${VERSION:${M_cmpv}} < ${3.1.12:L:${M_cmpv}}
```

Here `$_` is used to save the result of the `:S` modifier which is later referenced using the index values from `:range`.

:U*newval*

If the variable is undefined, the optional *newval* (which may be empty) is the value. If the variable is defined, the existing value is returned. This is another ODE make feature. It is handy for setting per-target CFLAGS for instance:

```
${_${.TARGET:T}_CFLAGS:U${DEF_CFLAGS}}
```

If a value is only required if the variable is undefined, use:

```
${VAR:D:Unewval}
```

:D*newval*

If the variable is defined, *newval* (which may be empty) is the value.

:L The name of the variable is the value.

:P The path of the node which has the same name as the variable is the value. If no such node exists or its path is null, the name of the variable is used. In order for this modifier to work, the name (node) must at least have appeared on the right-hand side of a dependency.

:!*cmd***!**

The output of running *cmd* is the value.

:sh The value is run as a command, and the output becomes the new value.

:sh1 The value is run as a command, for the first reference only, and the output is cached for subsequent references. This modifier is useful when the result is not expected to change.

::=*str*

The variable is assigned the value *str* after substitution. This modifier and its variations are useful in obscure situations such as wanting to set a variable at a point where a target's shell commands are being parsed. These assignment modifiers always expand to nothing.

The '**::**' helps avoid false matches with the AT&T System V UNIX style `:=` modifier and since substitution always occurs, the `::=` form is vaguely appropriate.

::?=*str*

As for **::=** but only if the variable does not already have a value.

::+=*str*
Append *str* to the variable.

::!=*cmd*
Assign the output of *cmd* to the variable.

:[*range***]**
Selects one or more words from the value, or performs other operations related to the way in which the value is split into words.

An empty value, or a value that consists entirely of white-space, is treated as a single word. For the purposes of the '**:[]**' modifier, the words are indexed both forwards using positive integers (where index 1 represents the first word), and backwards using negative integers (where index −1 represents the last word).

The *range* is subjected to variable expansion, and the expanded result is then interpreted as follows:

index Selects a single word from the value.

*start***..***end*
Selects all words from *start* to *end*, inclusive. For example, '**:[2..-1]**' selects all words from the second word to the last word. If *start* is greater than *end*, the words are output in reverse order. For example, '**:[-1..1]**' selects all the words from last to first. If the list is already ordered, this effectively reverses the list, but it is more efficient to use '**:Or**' instead of '**:O:[-1..1]**'.

∗ Causes subsequent modifiers to treat the value as a single word (possibly containing embedded whitespace). Analogous to the effect of **$**∗ in Bourne shell.

0 Means the same as '**:[**∗**]**'.

@ Causes subsequent modifiers to treat the value as a sequence of words delimited by whitespace. Analogous to the effect of **$@** in Bourne shell.

Returns the number of words in the value.

DIRECTIVES

make offers directives for including makefiles, conditionals and for loops. All these directives are identified by a line beginning with a single dot (**.**) character, followed by the keyword of the directive, such as **include** or **if**.

File inclusion

Files are included with either **.include <***file***>** or **.include "***file***"**. Variables between the angle brackets or double quotes are expanded to form the file name. If angle brackets are used, the included makefile is expected to be in the system makefile directory. If double quotes are used, the including makefile's directory and any directories specified using the **−I** option are searched before the system makefile directory.

For compatibility with other make variants, '**include** *file* ...' (without leading dot) is also accepted.

If the include statement is written as **.-include** or as **.sinclude**, errors locating and/or opening include files are ignored.

If the include statement is written as **.dinclude**, not only are errors locating and/or opening include files ignored, but stale dependencies within the included file are ignored just like in *.MAKE.DEPENDFILE*.

Exporting variables

The directives for exporting and unexporting variables are:

.export *variable* ...
Export the specified global variable.

For compatibility with other make programs, **export** *variable***=***value* (without leading dot) is also accepted.

Appending a variable name to *.MAKE.EXPORTED* is equivalent to exporting a variable.

.export-all
Export all globals except for internal variables (those that start with `.`). This is not affected by the **−X** flag, so should be used with caution.

.export-env *variable* ...
The same as `.export`, except that the variable is not appended to *.MAKE.EXPORTED*. This allows exporting a value to the environment which is different from that used by **make** internally.

.export-literal *variable* ...
The same as `.export-env`, except that variables in the value are not expanded.

.unexport *variable* ...
The opposite of `.export`. The specified global *variable* is removed from *.MAKE.EXPORTED*. If no variable list is provided, all globals are unexported, and *.MAKE.EXPORTED* deleted.

.unexport-env
Unexport all globals previously exported and clear the environment inherited from the parent. This operation causes a memory leak of the original environment, so should be used sparingly. Testing for *.MAKE.LEVEL* being 0 would make sense. Also note that any variables which originated in the parent environment should be explicitly preserved if desired. For example:

```
.if ${.MAKE.LEVEL} == 0
PATH := ${PATH}
.unexport-env
.export PATH
.endif
```

Would result in an environment containing only '*PATH*', which is the minimal useful environment. Actually '*.MAKE.LEVEL*' is also pushed into the new environment.

Messages
The directives for printing messages to the output are:

.info *message*
The message is printed along with the name of the makefile and line number.

.warning *message*
The message prefixed by '**warning:**' is printed along with the name of the makefile and line number.

.error *message*
The message is printed along with the name of the makefile and line number, **make** exits immediately.

Conditionals
The directives for conditionals are:

.if [**!**]*expression* [*operator expression* ...]
Test the value of an expression.

.ifdef [**!**]*variable* [*operator variable* ...]
Test whether a variable is defined.

.ifndef [**!**]*variable* [*operator variable* ...]
Test whether a variable is not defined.

.ifmake [**!**]*target* [*operator target* ...]
Test the target being requested.

.ifnmake [**!**]*target* [*operator target* ...]
Test the target being requested.

.else Reverse the sense of the last conditional.

.elif [**!**]*expression* [*operator expression* ...]
A combination of '**.else**' followed by '**.if**'.

.elifdef [**!**]*variable* [*operator variable* ...]
A combination of '**.else**' followed by '**.ifdef**'.

.elifndef [**!**]*variable* [*operator variable* ...]
A combination of '**.else**' followed by '**.ifndef**'.

.elifmake [**!**]*target* [*operator target* ...]
A combination of '**.else**' followed by '**.ifmake**'.

.elifnmake [**!**]*target* [*operator target* ...]
A combination of '**.else**' followed by '**.ifnmake**'.

.endif End the body of the conditional.

The *operator* may be any one of the following:

|| Logical OR.

&& Logical AND; of higher precedence than '**||**'.

make only evaluates a conditional as far as is necessary to determine its value. Parentheses can be used to override the operator precedence. The boolean operator '**!**' may be used to logically negate an expression, typically a function call. It is of higher precedence than '**&&**'.

The value of *expression* may be any of the following function call expressions:

defined(*varname***)**
Evaluates to true if the variable *varname* has been defined.

make(*target***)**
Evaluates to true if the target was specified as part of **make**'s command line or was declared the default target (either implicitly or explicitly, see *.MAIN*) before the line containing the conditional.

empty(*varname*[:*modifiers*]**)**
Evaluates to true if the expansion of the variable, after applying the modifiers, results in an empty string.

exists(*pathname***)**
Evaluates to true if the given pathname exists. If relative, the pathname is searched for on the system search path (see *.PATH*).

target(*target***)**
Evaluates to true if the target has been defined.

commands(*target***)**
Evaluates to true if the target has been defined and has commands associated with it.

Expression may also be an arithmetic or string comparison. Variable expansion is performed on both sides of the comparison. If both sides are numeric and neither is enclosed in quotes, the comparison is done numerically, otherwise lexicographically. A string is interpreted as a hexadecimal integer if it is preceded by **0x**, otherwise it is interpreted as a decimal floating-point number; octal numbers are not supported.

All comparisons may use the operators '**==**' and '**!=**'. Numeric comparisons may also use the operators '**<**', '**<=**', '**>**' and '**>=**'.

If the comparison has neither a comparison operator nor a right side, the expression evaluates to true if it is nonempty and its numeric value (if any) is not zero.

When **make** is evaluating one of these conditional expressions, and it encounters a (whitespace-separated) word it doesn't recognize, either the "make" or "defined" function is applied to it, depending on the form of the conditional. If the form is '**.ifdef**', '**.ifndef**' or '**.if**', the "defined" function is applied. Similarly, if the form is '**.ifmake**' or '**.ifnmake**', the "make" function is applied.

If the conditional evaluates to true, parsing of the makefile continues as before. If it evaluates to false, the following lines until the corresponding '**.elif**' variant, '**.else**' or '**.endif**' are skipped.

For loops

For loops are typically used to apply a set of rules to a list of files. The syntax of a for loop is:

.for *variable* [*variable* ...] **in** *expression*
⟨*make-lines*⟩
.endfor

The *expression* is expanded and then split into words. On each iteration of the loop, one word is taken and assigned to each *variable*, in order, and these *variables* are substituted into the *make-lines* inside the body of the for loop. The number of words must come out even; that is, if there are three iteration variables, the number of words provided must be a multiple of three.

If '**.break**' is encountered within a **.for** loop, it causes early termination of the loop, otherwise a parse error.

Other directives

.undef *variable* ...
Un-define the specified global variables. Only global variables can be un-defined.

COMMENTS

Comments begin with a hash (#) character, anywhere but in a shell command line, and continue to the end of an unescaped new line.

SPECIAL SOURCES (ATTRIBUTES)

.EXEC Target is never out of date, but always execute commands anyway.

.IGNORE Ignore any errors from the commands associated with this target, exactly as if they all were preceded by a dash (−).

.MADE Mark all sources of this target as being up to date.

.MAKE Execute the commands associated with this target even if the **−n** or **−t** options were specified. Normally used to mark recursive **make**s.

.META Create a meta file for the target, even if it is flagged as **.PHONY**, **.MAKE**, or **.SPECIAL**. Usage in conjunction with **.MAKE** is the most likely case. In "meta" mode, the target is out-of-date if the meta file is missing.

.NOMETA Do not create a meta file for the target. Meta files are also not created for **.PHONY**, **.MAKE**, or **.SPECIAL** targets.

.NOMETA_CMP
Ignore differences in commands when deciding if target is out of date. This is useful if the command contains a value which always changes. If the number of commands change, though, the target is still considered out of date. The same effect applies to any command line that uses the variable *.OODATE*, which can be used for that purpose even when not otherwise needed or desired:

```
skip-compare-for-some:
        @echo this is compared
        @echo this is not ${.OODATE:M.NOMETA_CMP}
```

```
@echo this is also compared
```

The **:M** pattern suppresses any expansion of the unwanted variable.

.NOPATH Do not search for the target in the directories specified by *.PATH*.

.NOTMAIN
Normally **make** selects the first target it encounters as the default target to be built if no target was specified. This source prevents this target from being selected.

.OPTIONAL
If a target is marked with this attribute and **make** can't figure out how to create it, it ignores this fact and assumes the file isn't needed or already exists.

.PHONY The target does not correspond to an actual file; it is always considered to be out of date, and is not created with the **−t** option. Suffix-transformation rules are not applied to **.PHONY** targets.

.PRECIOUS
When **make** is interrupted, it normally removes any partially made targets. This source prevents the target from being removed.

.RECURSIVE
Synonym for **.MAKE**.

.SILENT Do not echo any of the commands associated with this target, exactly as if they all were preceded by an at sign (`@`).

.USE Turn the target into **make**'s version of a macro. When the target is used as a source for another target, the other target acquires the commands, sources, and attributes (except for **.USE**) of the source. If the target already has commands, the **.USE** target's commands are appended to them.

.USEBEFORE
Like **.USE**, but instead of appending, prepend the **.USEBEFORE** target commands to the target.

.WAIT If **.WAIT** appears in a dependency line, the sources that precede it are made before the sources that succeed it in the line. Since the dependents of files are not made until the file itself could be made, this also stops the dependents being built unless they are needed for another branch of the dependency tree. So given:

```
x: a .WAIT b
        echo x
a:
        echo a
b: b1
        echo b
b1:
        echo b1
```

the output is always `a`, `b1`, `b`, `x`.

The ordering imposed by **.WAIT** is only relevant for parallel makes.

SPECIAL TARGETS

Special targets may not be included with other targets, i.e. they must be the only target specified.

.BEGIN Any command lines attached to this target are executed before anything else is done.

.DEFAULT
This is sort of a **.USE** rule for any target (that was used only as a source) that **make** can't figure out any other way to create. Only the shell script is used. The *.IMPSRC* variable of a target that inherits **.DEFAULT**'s commands is set to the target's own name.

.DELETE_ON_ERROR
If this target is present in the makefile, it globally causes make to delete targets whose commands fail. (By default, only targets whose commands are interrupted during execution are deleted. This is the historical behavior.) This setting can be used to help prevent half-finished or malformed targets from being left around and corrupting future rebuilds.

.END Any command lines attached to this target are executed after everything else is done successfully.

.ERROR Any command lines attached to this target are executed when another target fails. See *MAKE_PRINT_VAR_ON_ERROR* for the variables that will be set.

.IGNORE
Mark each of the sources with the **.IGNORE** attribute. If no sources are specified, this is the equivalent of specifying the **−i** option.

.INTERRUPT
If **make** is interrupted, the commands for this target are executed.

.MAIN If no target is specified when **make** is invoked, this target is built.

.MAKEFLAGS
This target provides a way to specify flags for **make** at the time when the makefiles are read. The flags are as if typed to the shell, though the **−f** option has no effect.

.NOPATH
Apply the **.NOPATH** attribute to any specified sources.

.NOTPARALLEL
Disable parallel mode.

.NO_PARALLEL
Synonym for **.NOTPARALLEL**, for compatibility with other pmake variants.

.NOREADONLY
clear the read-only attribute from the global variables specified as sources.

.OBJDIR The source is a new value for '*.OBJDIR*'. If it exists, **make** changes the current working directory to it and updates the value of '*.OBJDIR*'.

.ORDER In parallel mode, the named targets are made in sequence. This ordering does not add targets to the list of targets to be made.

Since the dependents of a target do not get built until the target itself could be built, unless `a` is built by another part of the dependency graph, the following is a dependency loop:

```
.ORDER: b a
b: a
```

.PATH The sources are directories which are to be searched for files not found in the current directory. If no sources are specified, any previously specified directories are removed from the search path. If the source is the special **.DOTLAST** target, the current working directory is searched last.

.PATH.*suffix*
Like **.PATH** but applies only to files with a particular suffix. The suffix must have been previously declared with **.SUFFIXES**.

.PHONY Apply the **.PHONY** attribute to any specified sources.

.POSIX If this is the first non-comment line in the main makefile, the variable *%POSIX* is set to the value `1003.2` and the makefile `<posix.mk>` is included if it exists, to provide POSIX-compatible default rules. If **make** is run with the **−r** flag, only `posix.mk` contributes to the default rules. In POSIX-compatible mode, the AT&T System V UNIX style substitution modifier is checked first

rather than as a fallback.

.PRECIOUS
Apply the **.PRECIOUS** attribute to any specified sources. If no sources are specified, the **.PRECIOUS** attribute is applied to every target in the file.

.READONLY
set the read-only attribute on the global variables specified as sources.

.SHELL Sets the shell that **make** uses to execute commands. The sources are a set of *field*=*value* pairs.

name This is the minimal specification, used to select one of the built-in shell specs; **sh**, **ksh**, and **csh**.

path Specifies the absolute path to the shell.

hasErrCtl Indicates whether the shell supports exit on error.

check The command to turn on error checking.

ignore The command to disable error checking.

echo The command to turn on echoing of commands executed.

quiet The command to turn off echoing of commands executed.

filter The output to filter after issuing the **quiet** command. It is typically identical to **quiet**.

errFlag The flag to pass the shell to enable error checking.

echoFlag The flag to pass the shell to enable command echoing.

newline The string literal to pass the shell that results in a single newline character when used outside of any quoting characters.

Example:

```
.SHELL: name=ksh path=/bin/ksh hasErrCtl=true \
        check="set -e" ignore="set +e" \
        echo="set -v" quiet="set +v" filter="set +v" \
        echoFlag=v errFlag=e newline="'\n'"
```

.SILENT Apply the **.SILENT** attribute to any specified sources. If no sources are specified, the **.SILENT** attribute is applied to every command in the file.

.STALE This target gets run when a dependency file contains stale entries, having *.ALLSRC* set to the name of that dependency file.

.SUFFIXES
Each source specifies a suffix to **make**. If no sources are specified, any previously specified suffixes are deleted. It allows the creation of suffix-transformation rules.

Example:

```
.SUFFIXES: .c .o
.c.o:
        cc -o ${.TARGET} -c ${.IMPSRC}
```

.SYSPATH
The sources are directories which are to be added to the system include path which **make** searches for makefiles. If no sources are specified, any previously specified directories are removed from the system include path.

ENVIRONMENT

make uses the following environment variables, if they exist: *MACHINE*, *MACHINE_ARCH*, *MAKE*, *MAKEFLAGS*, *MAKEOBJDIR*, *MAKEOBJDIRPREFIX*, *MAKESYSPATH*, *MAKE_STACK_TRACE*, *PWD*, and *TMPDIR*.

MAKEOBJDIRPREFIX and *MAKEOBJDIR* should be set in the environment or on the command line to **make** and not as makefile variables; see the description of '.*OBJDIR*' for more details. It is possible to set these via makefile variables but unless done very early and the '**.OBJDIR**' target is used to reset '.*OBJDIR*', there may be unexpected side effects.

If the *MAKE_STACK_TRACE* environment variable is set to "yes", any stack traces include the call chain of the parent processes.

FILES

.depend	list of dependencies
makefile	first default makefile if no makefile is specified on the command line
Makefile	second default makefile if no makefile is specified on the command line
sys.mk	system makefile
/usr/share/mk	system makefile directory

DIAGNOSTICS

Invalid internal option "-J" in "*directory*"

The internal **−J** option coordinates the main **make** process with the sub-make processes to limit the number of jobs that run in parallel. The option is passed to all child processes via the *MAKEFLAGS* environment variable. To become valid, this option requires that the target running the sub-make is marked with the .MAKE special source, or that one of the target's commands directly contains the word "make" or one of the expressions "${MAKE}", "${.MAKE}", "$(MAKE)", "$(.MAKE)". If that's not the case, make issues the above warning and falls back to compat mode.

To see the chain of sub-makes that leads to the invalid option, set the *MAKE_STACK_TRACE* environment variable to "yes".

To run the sub-make in parallel mode, even in dry-run mode (see the **−n** option), add the .MAKE pseudo source to the target. This is appropriate when the sub-make runs the same target in a subdirectory.

To run the sub-make in parallel mode but not in dry-mode, add a "${:D make}" marker to one of the target's commands. This marker expands to an empty string and thus does not affect the executed commands.

To run the sub-make in compat mode, add the **−B** option to its invocation. This is appropriate when the sub-make is only used to print a variable's value using the **−v** or **−V** options.

To make the sub-make independent from the parent make, unset the *MAKEFLAGS* environment variable in the target's commands.

COMPATIBILITY

The basic make syntax is compatible between different make variants; however the special variables, variable modifiers and conditionals are not.

Older versions

An incomplete list of changes in older versions of **make**:

The way that .for loop variables are substituted changed after NetBSD 5.0 so that they still appear to be variable expansions. In particular this stops them being treated as syntax, and removes some obscure problems using them in .if statements.

The way that parallel makes are scheduled changed in NetBSD 4.0 so that .ORDER and .WAIT apply recursively to the dependent nodes. The algorithms used may change again in the future.

Other make dialects

Other make dialects (GNU make, SVR4 make, POSIX make, etc.) do not support most of the features of **make** as described in this manual. Most notably:

- The **.WAIT** and **.ORDER** declarations and most functionality pertaining to parallelization. (GNU make supports parallelization but lacks the features needed to control it effectively.)
- Directives, including for loops and conditionals and most of the forms of include files. (GNU make has its own incompatible and less powerful syntax for conditionals.)
- All built-in variables that begin with a dot.
- Most of the special sources and targets that begin with a dot, with the notable exception of **.PHONY**, **.PRECIOUS**, and **.SUFFIXES**.
- Variable modifiers, except for the `:old=new` string substitution, which does not portably support globbing with `%` and historically only works on declared suffixes.
- The **$>** variable even in its short form; most makes support this functionality but its name varies.

Some features are somewhat more portable, such as assignment with **+=**, **?=**, and **!=**. The *.PATH* functionality is based on an older feature **VPATH** found in GNU make and many versions of SVR4 make; however, historically its behavior is too ill-defined (and too buggy) to rely upon.

The **$@** and **$<** variables are more or less universally portable, as is the **$(MAKE)** variable. Basic use of suffix rules (for files only in the current directory, not trying to chain transformations together, etc.) is also reasonably portable.

SEE ALSO

mkdep(1), *style.Makefile*(5)

HISTORY

A **make** command appeared in Version 7 AT&T UNIX. This **make** implementation is based on Adam de Boor's pmake program, which was written for Sprite at Berkeley. It was designed to be a parallel distributed make running jobs on different machines using a daemon called "customs".

Historically the target/dependency **FRC** has been used to FoRCe rebuilding (since the target/dependency does not exist ... unless someone creates an *FRC* file).

BUGS

The **make** syntax is difficult to parse. For instance, finding the end of a variable's use should involve scanning each of the modifiers, using the correct terminator for each field. In many places **make** just counts {} and () in order to find the end of a variable expansion.

There is no way of escaping a space character in a filename.

In jobs mode, when a target fails; **make** will put an error token into the job token pool. This will cause all other instances of **make** using that token pool to abort the build and exit with error code 6. Sometimes the attempt to suppress a cascade of unnecessary errors, can result in a seemingly unexplained `*** Error code 6`