NAME

make – maintain, update, and regenerate related programs and files

SYNOPSIS

/usr/ccs/bin/make [–d] [–dd] [–D] [–DD] [–e] [–i] [–k] [–n] [–p]
[–P] [–q] [–r] [–s] [–S] [–t] [–V] [–f *makefile* **]** . . . **[–K** *statefile* **]** . . .
[*target* . . .] [*macro***=***value* . . .]

AVAILABILITY

SUNWsprot

DESCRIPTION

make executes a list of shell commands associated with each *target*, typically to create or update a file of the same name. *makefile* contains entries that describe how to bring a target up to date with respect to those on which it depends, which are called *dependencies*. Since each dependency is a target, it may have dependencies of its own. Targets, dependencies, and sub-dependencies comprise a tree structure that **make** traces when deciding whether or not to rebuild a *target*.

make recursively checks each *target* against its dependencies, beginning with the first target entry in *makefile* if no *target* argument is supplied on the command line. If, after processing all of its dependencies, a target file is found either to be missing, or to be older than any of its dependencies, **make** rebuilds it. Optionally with this version of **make**, a target can be treated as out-of-date when the commands used to generate it have changed since the last time the target was built.

To build a given target, **make** executes the list of commands, called a *rule*. This rule may be listed explicitly in the target's makefile entry, or it may be supplied implicitly by **make**.

Except when in POSIX mode, when no *makefile* is specified with a **–f** option:

- If there is a file named **makefile** in the working directory, **make** uses that file. If, however, there is an SCCS history file (**SCCS/s.makefile**) which is newer, **make** attempts to retrieve and use the most recent version.
- In the absence of the above file(s), if a file named **Makefile** is present in the working directory, **make** attempts to use it. If there is an SCCS history file (**SCCS/s.Makefile**) that is newer, **make** attempts to retrieve and use the most recent version.

In POSIX mode, when no *makefile* is specified with a **–f** option, **make** tries the following files in sequence:

- **./makefile**, **./Makefile**
- **s.makefile**, **SCCS/s.makefile**
- **s.Makefile**, **SCCS/s.Makefile**

If no *target* is specified on the command line, **make** uses the first target defined in *makefile*.

If a *target* has no makefile entry, or if its entry has no rule, **make** attempts to derive a rule by each of the following methods, in turn, until a suitable rule is found. (Each method is described under **USAGE** below.)

- Pattern matching rules.
- Implicit rules, read in from a user-supplied makefile.

- Standard implicit rules (also known as suffix rules), typically read in from the file **/usr/share/lib/make/make.rules**.
- SCCS retrieval. **make** retrieves the most recent version from the SCCS history file (if any). See the description of the **.SCCS_GET:** special-function target for details.
- The rule from the **.DEFAULT:** target entry, if there is such an entry in the makefile.

If there is no makefile entry for a *target*, if no rule can be derived for building it, and if no file by that name is present, **make** issues an error message and halts.

OPTIONS

The following options are supported:

–d Display the reasons why **make** chooses to rebuild a target; **make** displays any and all dependencies that are newer. In addition, **make** displays options read in from the **MAKEFLAGS** environment variable.

–dd Display the dependency check and processing in vast detail.

–D Display the text of the makefiles read in.

–DD Display the text of the makefiles, **make.rules** file, the state file, and all hidden-dependency reports.

–e Environment variables override assignments within makefiles.

–f *makefile* Use the description file *makefile*. A '–' as the *makefile* argument denotes the standard input. The contents of *makefile*, when present, override the standard set of implicit rules and predefined macros. When more than one '**–f** *makefile*' argument pair appears, **make** uses the concatenation of those files, in order of appearance.

–i Ignore error codes returned by commands. Equivalent to the special-function target '**.IGNORE:**'.

–k When a nonzero error status is returned by a rule, or when **make** cannot find a rule, abandon work on the current target, but continue with other dependency branches that do not depend on it.

–K *statefile* Use the state file *statefile*. A '–' as the *statefile* argument denotes the standard input. The contents of *statefile*, when present, override the standard set of implicit rules and predefined macros. When more than one '**–K** *statefile*' argument pair appears, **make** uses the concatenation of those files, in order of appearance. (See also **.KEEP_STATE** and **.KEEP_STATE_FILE** in the **Special-Functions Targets** section).

–n No execution mode. Print commands, but do not execute them. Even lines beginning with an **@** are printed. However, if a command line contains a reference to the **$(MAKE)** macro, that line is always executed (see the discussion of **MAKEFLAGS** in **Reading Makefiles and the Environment**). When **.POSIX** is in effect, lines beginning with a "+" are executed.

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

−P Merely report dependencies, rather than building them.

−q Question mode. **make** returns a zero or nonzero status code depending on whether or not the target file is up to date. When **.POSIX** is in effect, lines beginning with a "+" are executed.

−r Do not read in the default makefile **/usr/share/lib/make/make.rules**.

−s Silent mode. Do not print command lines before executing them. Equivalent to the special-function target **.SILENT:**.

−S Undo the effect of the **−k** option. Stop processing when a non-zero exit status is returned by a command.

−t Touch the target files (bringing them up to date) rather than performing their rules. *This can be dangerous when files are maintained by more than one person.* When the **.KEEP_STATE:** target appears in the makefile, this option updates the state file just as if the rules had been performed. When **.POSIX** is in effect, lines beginning with a "+" are executed.

−V Puts **make** into SysV mode. Refer to **sysV-make**(1) for respective details.

OPERANDS

The following operands are supported:

target Target names, as defined in **USAGE**.

*macro***=***value*
Macro definition. This definition overrides any regular definition for the specified macro within the makefile itself, or in the environment. However, this definition can still be overridden by conditional macro assignments.

USAGE

Refer to **make** in *Programming Utilities Guide* for tutorial information.

Reading Makefiles and the Environment

When **make** first starts, it reads the **MAKEFLAGS** environment variable to obtain any the following options specified present in its value: −**d**, −**D**, −**e**, −**i**, −**k**, −**n**, −**p**, −**q**, −**r**, −**s**, −**S**, or −**t**. Due to the implementation of POSIX.2 standardization, the **MAKEFLAGS** values will contain a leading '**—**' character. **make** then reads the command line for additional options, which also take effect.

Next, **make** reads in a default makefile that typically contains predefined macro definitions, target entries for implicit rules, and additional rules, such as the rule for retrieving SCCS files. If present, **make** uses the file **make.rules** in the current directory; otherwise it reads the file **/usr/share/lib/make/make.rules**, which contains the standard definitions and rules.
Use the directive:

include /usr/share/lib/make/make.rules

in your local **make.rules** file to include them.

Next, **make** imports variables from the environment (unless the –**e** option is in effect), and treats them as defined macros. Because **make** uses the most recent definition it encounters, a macro definition in the makefile normally overrides an environment variable of the same name. When –**e** is in effect, however, environment variables are read in *after* all makefiles have been read. In that case, the environment variables take precedence over definitions in the makefile.

Next, **make** reads any makefiles you specify with –**f**, or one of **makefile** or **Makefile** as described above and then the state file, in the local directory if it exists. If the makefile contains a **.KEEP_STATE_FILE** target, then it reads the state file that follows the target. Refer to special target **.KEEP_STATE_FILE** for details.

Next, (after reading the environment if –**e** is in effect), **make** reads in any macro definitions supplied as command line arguments. These override macro definitions in the makefile and the environment both, but only for the **make** command itself.

make exports environment variables, using the most recently defined value. Macro definitions supplied on the command line are not normally exported, unless the macro is also an environment variable.

make does not export macros defined in the makefile. If an environment variable is set, and a macro with the same name is defined on the command line, **make** exports its value as defined on the command line. Unless –**e** is in effect, macro definitions within the makefile take precedence over those imported from the environment.

The macros **MAKEFLAGS**, **MAKE**, **SHELL**, **HOST_ARCH**, **HOST_MACH**, and **TARGET_MACH** are special cases. See **Special-Purpose Macros**, below for details.

Makefile Target Entries

A target entry has the following format:

target. . . [**:** | **::**] [*dependency*] . . . [**;** *command*] . . .
[*command*]
. . .

The first line contains the name of a target, or a space-separated list of target names, terminated with a colon or double colon. If a list of targets is given, this is equivalent to having a separate entry of the same form for each target. The colon(s) may be followed by a *dependency*, or a dependency list. **make** checks this list before building the target. The dependency list may be terminated with a semicolon (**;**), which in turn can be followed by a single Bourne shell command. Subsequent lines in the target entry begin with a TAB, and contain Bourne shell commands. These commands comprise the rule for building the target.

Shell commands may be continued across input lines by escaping the NEWLINE with a backslash (\). The continuing line must also start with a TAB.

To rebuild a target, **make** expands macros, strips off initial TAB characters and either executes the command directly (if it contains no shell metacharacters), or passes each command line to a Bourne shell for execution.

The first line that does not begin with a TAB or '**#**' begins another target or macro definition.

Special Characters

Global

Start a comment. The comment ends at the next NEWLINE. If the '**#**' follows the TAB in a command line, that line is passed to the shell (which also treats '**#**' as the start of a comment).

include *filename* If the word **include** appears as the first seven letters of a line and is followed by a SPACE or TAB, the string that follows is taken as a filename to interpolate at that line. **include** files can be nested to a depth of no more than about 16. If *filename* is a macro reference, it is expanded.

Targets and Dependencies

: Target list terminator. Words following the colon are added to the dependency list for the target or targets. If a target is named in more than one colon-terminated target entry, the dependencies for all its entries are added to form that target's complete dependency list.

:: Target terminator for alternate dependencies. When used in place of a '**:**' the double-colon allows a target to be checked and updated with respect to alternate dependency lists. When the target is out-of-date with respect to dependencies listed in the first alternate, it is built according to the rule for that entry. When out-of-date with respect to dependencies in another alternate, it is built according the rule in that other entry. Implicit rules do not apply to double-colon targets; you must supply a rule for each entry. If no dependencies are specified, the rule is always performed.

target [**+** *target*. . .] **:**
Target group. The rule in the target entry builds all the indicated targets as a group. It is normally performed only once per **make** run, but is checked for command dependencies every time a target in the group is encountered in the dependency scan.

% Pattern matching wild card metacharacter. Like the '*' shell wild card, '**%**' matches any string of zero or more characters in a target name or dependency, in the target portion of a conditional macro definition, or within a pattern replacement macro reference. Note that only one '**%**' can appear in a target, dependency-name, or pattern-replacement macro reference.

./*pathname* **make** ignores the leading '**./**' characters from targets with names given as pathnames relative to "dot," the working directory.

Macros

= Macro definition. The word to the left of this character is the macro name; words to the right comprise its value. Leading and trailing white space characters are stripped from the value. A word break following the **=** is implied.

$ Macro reference. The following character, or the parenthesized or bracketed string, is interpreted as a macro reference: **make** expands the reference (including the **$**) by replacing it with the macro's value.

( )
{ } Macro-reference name delimiters. A parenthesized or bracketed word appended to a **$** is taken as the name of the macro being referred to. Without the delimiters, **make** recognizes only the first character as the macro name.

$$** A reference to the dollar-sign macro, the value of which is the character '**$'. Used to pass variable expressions beginning with **$** to the shell, to refer to environment variables which are expanded by the shell, or to delay processing of dynamic macros within the dependency list of a target, until that target is actually processed.

\$ Escaped dollar-sign character. Interpreted as a literal dollar sign within a rule.

+= When used in place of '**=**', appends a string to a macro definition (must be surrounded by white space, unlike '**=**').

:= Conditional macro assignment. When preceded by a list of targets with explicit target entries, the macro definition that follows takes effect when processing only those targets, and their dependencies.

:sh = Define the value of a macro to be the output of a command (see **Command Substitutions**, below).

:sh In a macro reference, execute the command stored in the macro, and replace the reference with the output of that command (see **Command Substitutions**).

Rules

\+ **make** will always execute the commands preceded by a **"+"**, even when −**n** is specified.

− **make** ignores any nonzero error code returned by a command line for which the first non-TAB character is a '−'. This character is not passed to the shell as part of the command line. **make** normally terminates when a command returns nonzero status, unless the −**i** or −**k** options, or the **.IGNORE:** special-function target is in effect.

@ If the first non-TAB character is a **@**, **make** does not print the command line before executing it. This character is not passed to the shell.

? Escape command-dependency checking. Command lines starting with this character are not subject to command dependency checking.

! Force command-dependency checking. Command-dependency checking is applied to command lines for which it would otherwise be suppressed. This checking is normally suppressed for lines that contain references to the '**?**' dynamic macro (for example, '**$?**').

When any combination of '+', '−', '@', '**?**', or '**!**' appear as the first characters after the TAB, all that are present apply. None are passed to the shell.

Special-Function Targets

When incorporated in a makefile, the following target names perform special-functions:

.DEFAULT: If it has an entry in the makefile, the rule for this target is used to process a target when there is no other entry for it, no rule for building it, and no SCCS history file from which to retrieve a current version. **make** ignores any dependencies for this target.

.DONE: If defined in the makefile, **make** processes this target and its dependencies after all other targets are built. This target is also performed when **make** halts with an error, unless the **.FAILED** target is defined.

.FAILED: This target, along with its dependencies, is performed instead of **.DONE** when defined in the makefile and **make** halts with an error.

.GET_POSIX: This target contains the rule for retrieving the current version of an SCCS file from its history file in the current working directory. **make** uses this rule when it is running in POSIX mode.

.IGNORE: Ignore errors. When this target appears in the makefile, **make** ignores non-zero error codes returned from commands. When used under POSIX mode, **.IGNORE** could be followed by target names only, for which the errors will be ignored.

.INIT: If defined in the makefile, this target and its dependencies are built before any other targets are processed.

.KEEP_STATE: If this target is in effect, **make** updates the state file, **.make.state**, in the current directory. This target also activates command dependencies, and hidden dependency checks. If either the **.KEEP_STATE:** target appears in the makefile, or the environment variable **KEEP_STATE** is set ("**setenv KEEP_STATE**"), **make** will rebuild everything in order to collect dependency information, even if all the targets were up to date due to previous **make** runs. (See also the **ENVIRONMENT** section.) This target has no effect if used under POSIX mode.

.KEEP_STATE_FILE:
This target has no effect if used under POSIX mode. This target implies **.KEEP_STATE**. If the target is followed by a filename, **make** uses it as the state file. If the target is followed by a directory name, **make** looks for a **.make.state** in that directory. If the target is not followed by any name, **make** looks for **.make.state** file in the current working directory.

.MAKE_VERSION:
A target-entry of the form:

.MAKE_VERSION: **VERSION**–*number*

enables version checking. If the version of **make** differs from the version indicated, **make** issues a warning message.

.NO_PARALLEL:
Currently, this target has no effect, it is, however, reserved for future

use.

.PARALLEL: Currently of no effect, but reserved for future use.

.POSIX: This target enables the POSIX compliant mode.

.PRECIOUS: List of files not to delete. **make** does not remove any of the files listed as dependencies for this target when interrupted. **make** normally removes the current target when it receives an interrupt. When used under POSIX mode, if the target is not followed by a list of files, all the file are assumed precious.

.SCCS_GET: This target contains the rule for retrieving the current version of an SCCS file from its history file. To suppress automatic retrieval, add an entry for this target with an empty rule to your makefile.

.SCCS_GET_POSIX:
This target contains the rule for retrieving the current version of an SCCS file from its history file. **make** uses this rule when it is running in POSIX mode.

.SILENT: Run silently. When this target appears in the makefile, **make** does not echo commands before executing them. When used in POSIX mode, it could be followed by target names, and only those will be executed silently.

.SUFFIXES: The suffixes list for selecting implicit rules (see **The Suffixes List**).

.WAIT: Currently of no effect, but reserved for future use.

Clearing Special Targets

In this version of **make**, you can clear the definition of the following special targets by supplying entries for them with no dependencies and no rule:

.DEFAULT, **.SCCS_GET**, and **.SUFFIXES**

Command Dependencies

When the **.KEEP_STATE:** target is effective, **make** checks the command for building a target against the state file. If the command has changed since the last **make** run, **make** rebuilds the target.

Hidden Dependencies

When the **.KEEP_STATE:** target is effective, **make** reads reports from **cpp**(1) and other compilation processors for any "hidden" files, such as **#include** files. If the target is out of date with respect to any of these files, **make** rebuilds it.

Macros

Entries of the form

macro=*value*

define macros. *macro* is the name of the macro, and *value*, which consists of all characters up to a comment character or unescaped NEWLINE, is the value. **make** strips both leading and trailing white space in accepting the value.

Subsequent references to the macro, of the forms: **$(***name***)** or **${***name***}** are replaced by *value*. The parentheses or brackets can be omitted in a reference to a macro with a single-character name.

Macro references can contain references to other macros, in which case nested references are expanded first.

Suffix Replacement Macro References

Substitutions within macros can be made as follows:

$(*name***:***string1***=***string2)*

where *string1* is either a suffix, or a word to be replaced in the macro definition, and *string2* is the replacement suffix or word. Words in a macro value are separated by SPACE, TAB, and escaped NEWLINE characters.

Pattern Replacement Macro References

Pattern matching replacements can also be applied to macros, with a reference of the form:

$(*name***:** *op***%***os***=** *np***%***ns***)**

where *op* is the existing (old) prefix and *os* is the existing (old) suffix, *np* and *ns* are the new prefix and new suffix, respectively, and the pattern matched by **%** (a string of zero or more characters), is carried forward from the value being replaced. For example:

PROGRAM=fabricate
DEBUG= $(PROGRAM:%=tmp/%−g)

sets the value of **DEBUG** to **tmp/fabricate−g**.

Note that pattern replacement macro references cannot be used in the dependency list of a pattern matching rule; the **%** characters are not evaluated independently. Also, any number of **%** metacharacters can appear after the equal-sign.

Appending to a Macro

Words can be appended to macro values as follows:

macro **+=** *word . . .*

Special-Purpose Macros

When the **MAKEFLAGS** variable is present in the environment, **make** takes options from it, in combination with options entered on the command line. **make** retains this combined value as the **MAKEFLAGS** macro, and exports it automatically to each command or shell it invokes.

Note that flags passed by way of **MAKEFLAGS** are only displayed when the −**d**, or −**dd** options are in effect.

The **MAKE** macro is another special case. It has the value **make** by default, and temporarily overrides the −**n** option for any line in which it is referred to. This allows nested invocations of **make** written as:

$(MAKE) . . .

to run recursively, with the −**n** flag in effect for all commands but **make**. This lets you use '**make** −**n**' to test an entire hierarchy of makefiles.

For compatibility with the 4.2 BSD **make**, the **MFLAGS** macro is set from the **MAKEFLAGS** variable by prepending a '−'. **MFLAGS** is not exported automatically.

The **SHELL** macro, when set to a single-word value such as **/usr/bin/csh**, indicates the name of an alternate shell to use. The default is **/bin/sh**. Note that **make** executes commands that contain no shell metacharacters itself. Built-in commands, such as **dirs** in the

C shell, are not recognized unless the command line includes a metacharacter (for instance, a semicolon). This macro is neither imported from, nor exported to the environment, regardless of **−e**. To be sure it is set properly, you must define this macro within every makefile that requires it.

The following macros are provided for use with cross-compilation:

HOST_ARCH The machine architecture of the host system. By default, this is the output of the **arch**(1) command prepended with '**—**'. Under normal circumstances, this value should never be altered by the user.

HOST_MACH The machine architecture of the host system. By default, this is the output of the **mach**(1), prepended with '**—**'. Under normal circumstances, this value should never be altered by the user.

TARGET_ARCH The machine architecture of the target system. By default, the output of **mach**, prepended with '**—**'.

Dynamic Macros

There are several dynamically maintained macros that are useful as abbreviations within rules. They are shown here as references; if you were to define them, **make** would simply override the definition.

$* The basename of the current target, derived as if selected for use with an implicit rule.

$< The name of a dependency file, derived as if selected for use with an implicit rule.

$@** The name of the current target. This is the only dynamic macro whose value is strictly determined when used in a dependency list. (In which case it takes the form '**$$@'.)

$? The list of dependencies that are newer than the target. Command-dependency checking is automatically suppressed for lines that contain this macro, just as if the command had been prefixed with a '**?**'. See the description of '**?**', under **Makefile Special Tokens**, above. You can force this check with the **!** command-line prefix.

$% The name of the library member being processed. (See **Library Maintenance**, below.)

To refer to the **$@** dynamic macro within a dependency list, precede the reference with an additional '**$**' character (as in, '**$$@**'). Because **make** assigns **$<** and **$*** as it would for implicit rules (according to the suffixes list and the directory contents), they may be unreliable when used within explicit target entries.

These macros can be modified to apply either to the filename part, or the directory part of the strings they stand for, by adding an upper case **F** or **D**, respectively (and enclosing the resulting name in parentheses or braces). Thus, '**$(@D)**' refers to the directory part of the string '**$@**'; if there is no directory part, '**.**' is assigned. **$(@F)** refers to the filename part.

Conditional Macro Definitions

A macro definition of the form:

target-list **:=** *macro* **=** *value*

indicates that when processing any of the targets listed *and their dependencies*, *macro* is to be set to the *value* supplied. Note that if a conditional macro is referred to in a dependency list, the **$** must be delayed (use **$$** instead). Also, *target-list* may contain a **%** pattern, in which case the macro will be conditionally defined for all targets encountered that match the pattern. A pattern replacement reference can be used within the *value*.

You can temporarily append to a macro's value with a conditional definition of the form:

target-list **:=** *macro* **+=** *value*

Predefined Macros

make supplies the macros shown in the table that follows for compilers and their options, host architectures, and other commands. Unless these macros are read in as environment variables, their values are not exported by **make**. If you run **make** with any of these set in the environment, it is a good idea to add commentary to the makefile to indicate what value each is expected to take. If −**r** is in effect, **make** does not read the default makefile (**./make.rules** or **/usr/share/lib/make/make.rules**) in which these macro definitions are supplied.

Table of Predefined Macros		
Use	*Macro*	*Default Value*
Library Archives	**AR** **ARFLAGS**	**ar** **rv**
Assembler Commands	**AS** **ASFLAGS** **COMPILE.s** **COMPILE.S**	**as** **$(AS) $(ASFLAGS)** **$(CC) $(ASFLAGS) $(CPPFLAGS) −c**
C Compiler Commands	**CC** **CFLAGS** **CPPFLAGS** **COMPILE.c** **LINK.c**	**cc** **$(CC) $(CFLAGS) $(CPPFLAGS) −c** **$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)**
C++ Compiler Commands	**CCC** **CCFLAGS** **CPPFLAGS** **COMPILE.cc** **LINK.cc** **COMPILE.C** **LINK.C**	**CC** **CFLAGS** **$(CCC) $(CCFLAGS) $(CPPFLAGS) −c** **$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)** **$(CCC) $(CCFLAGS) $(CPPFLAGS) −c** **$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)**
FORTRAN 77 Compiler Commands	**FC** **FFLAGS** **COMPILE.f** **LINK.f** **COMPILE.F** **LINK.F**	**f77** **$(FC) $(FFLAGS) −c** **$(FC) $(FFLAGS) $(LDFLAGS)** **$(FC) $(FFLAGS) $(CPPFLAGS) −c** **$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS)**
FORTRAN 90 Compiler Commands	**FC** **F90FLAGS** **COMPILE.f90** **LINK.f90** **COMPILE.ftn** **LINK.ftn**	**f90** **$(F90C) $(F90FLAGS) −c** **$(F90C)** **$(F90C) $(F90FLAGS) $(CPPFLAGS) −c** **$(F90C) $(F90FLAGS) $(CPPFLAGS) $(LDFLAGS)**
Link Editor Command	**LD** **LDFLAGS**	**ld**
lex Command	**LEX** **LFLAGS** **LEX.l**	**lex** **$(LEX) $(LFLAGS) −t**
lint Command	**LINT** **LINTFLAGS** **LINT.c**	**lint** **$(LINT) $(LINTFLAGS) $(CPPFLAGS)**
Modula 2 Commands	**M2C** **M2FLAGS** **MODFLAGS** **DEFFLAGS** **COMPILE.def** **COMPILE.mod**	**m2c** **$(M2C) $(M2FLAGS) $(DEFFLAGS)** **$(M2C) $(M2FLAGS) $(MODFLAGS)**

Table of Predefined Macros		
Use	*Macro*	*Default Value*
Pascal Compiler Commands	**PC** **PFLAGS** **COMPILE.p** **LINK.p**	**pc** **$(PC) $(PFLAGS) $(CPPFLAGS) −c** **$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS)**
Ratfor Compilation Commands	**RFLAGS** **COMPILE.r** **LINK.r**	 **$(FC) $(FFLAGS) $(RFLAGS) −c** **$(FC) $(FFLAGS) $(RFLAGS) $(LDFLAGS)**
rm Command	**RM**	**rm −f**
sccs Command	**SCCSFLAGS** **SCCSGETFLAGS**	 **−s**
yacc Command	**YACC** **YFLAGS** **YACC.y**	**yacc** **$(YACC) $(YFLAGS)**
Suffixes List	**SUFFIXES**	**.o .c .c~ .cc .cc~ .y .y~ .l .l~ .s .s~ .sh .sh~** **.S .S~ .ln .h .h~ .f .f~ .F .F~ .mod .mod~** **.sym .def .def~ .p .p~ .r .r~ .cps .cps~ .C .C~** **.Y .Y~ .L .L .f90 .f90~ .ftn .ftn~**

Implicit Rules

When a target has no entry in the makefile, **make** attempts to determine its class (if any) and apply the rule for that class. An implicit rule describes how to build any target of a given class, from an associated dependency file. The class of a target can be determined either by a pattern, or by a suffix; the corresponding dependency file (with the same basename) from which such a target might be built. In addition to a predefined set of implicit rules, make allows you to define your own, either by pattern, or by suffix.

Pattern Matching Rules

A target entry of the form:

tp%*ts* : *dp*%*ds*
    rule

is a pattern matching rule, in which *tp* is a target prefix, *ts* is a target suffix, *dp* is a dependency prefix, and *ds* is a dependency suffix (any of which may be null). The '%' stands for a basename of zero or more characters that is matched in the target, and is used to construct the name of a dependency. When **make** encounters a match in its search for an implicit rule, it uses the rule in that target entry to build the target from the dependency file. Pattern-matching implicit rules typically make use of the **$@** and **$<** dynamic macros as placeholders for the target and dependency names. Other, regular dependencies may occur in the dependency list; however, none of the regular dependencies may contain '%'. An entry of the form:

tp %*ts* **:** [*dependency* . . .] *dp* %*ds* [*dependency* . . .]
    rule

is a valid pattern matching rule.

Suffix Rules

When no pattern matching rule applies, **make** checks the target name to see if it ends with a suffix in the known suffixes list. If so, **make** checks for any suffix rules, as well as a dependency file with same root and another recognized suffix, from which to build it.

The target entry for a suffix rule takes the form:

DsTs: *rule*

where *Ts* is the suffix of the target, *Ds* is the suffix of the dependency file, and *rule* is the rule for building a target in the class. Both *Ds* and *Ts* must appear in the suffixes list. (A suffix need not begin with a '**.**' to be recognized.)

A suffix rule with only one suffix describes how to build a target having a null (or no) suffix from a dependency file with the indicated suffix. For instance, the **.c** rule could be used to build an executable program named **file** from a C source file named '**file.c**'. If a target with a null suffix has an explicit dependency, **make** omits the search for a suffix rule.

Table of Standard Implicit (Suffix) Rules

Use	*Implicit Rule Name*	*Command Line*
Assembly	**.s.o**	**$(COMPILE.s) −o $@ $<**
Files	**.s.a**	**$(COMPILE.s) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.s~.o	**$(−s1GET) $(−s1GFLAGS) −p $< > $*.s** **$(−s1COMPILE.s) −o $@ $*.s**
	.S.o	**$(COMPILE.S) −o $@ $<**
	.S.a	**$(COMPILE.S) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.S~.o	**$(GET) $(GFLAGS) −p $< > $*.S** **$(COMPILE.S) −o $@ $*.S**
	.S~.a	**$(GET) $(GFLAGS) −p $< > $*.S** **$(COMPILE.S) −o $% $*.S** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**

Table of Standard Implicit (Suffix) Rules		
Use	*Implicit Rule Name*	*Command Line*
C Files	**.c**	**$(LINK.c) −o $@ $< $(LDLIBS)**
	.c.ln	**$(LINT.c) $(OUTPUT_OPTION) −i $<**
	.c.o	**$(COMPILE.c) $(OUTPUT_OPTION) $<**
	.c.a	**$(COMPILE.c) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.c~	**$(GET) $(GFLAGS) −p $< > $*.c** **$(CC) $(CFLAGS) $(LDFLAGS) −o $@ $*.c**
	.c~.o	**$(GET) $(GFLAGS) −p $< > $*.c** **$(CC) $(CFLAGS) −c $*.c**
	.c~.ln	**$(GET) $(GFLAGS) −p $< > $*.c** **$(LINT.c) $(OUTPUT_OPTION) −c $*.c**
	.c~.a	**$(GET) $(GFLAGS) −p $< > $*.c** **$(COMPILE.c) −o $% $*.c** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**

Table of Standard Implicit (Suffix) Rules		
Use	*Implicit Rule Name*	*Command Line*
C++ Files	**.cc**	**$(LINK.cc) −o $@ $< $(LDLIBS)**
	.cc.o	**$(COMPILE.cc) $(OUTPUT_OPTION) $<**
	.cc.a	**$(COMPILE.cc) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.cc~	**$(GET) $(GFLAGS) −p $< > $*.cc** **$(LINK.cc) −o $@ $*.cc $(LDLIBS)**
	.cc.o	**$(COMPILE.cc) $(OUTPUT_OPTION) $<**
	.cc~.o	**$(GET) $(GFLAGS) −p $< > $*.cc** **$(COMPILE.cc) $(OUTPUT_OPTION) $*.cc**
	.cc.a	**$(COMPILE.cc) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.cc~.a	**$(GET) $(GFLAGS) −p $< > $*.cc** **$(COMPILE.cc) −o $% $*.cc** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.C	**$(LINK.C) −o $@ $< $(LDLIBS)**
	.C~	**$(GET) $(GFLAGS) −p $< > $*.C** **$(LINK.C) −o $@ $*.C $(LDLIBS)**
	.C.o	**$(COMPILE.C) $(OUTPUT_OPTION) $<**
	.C~.o	**$(GET) $(GFLAGS) −p $< > $*.C** **$(COMPILE.C) $(OUTPUT_OPTION) $*.C**
	.C.a	**$(COMPILE.C) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.C~.a	**$(GET) $(GFLAGS) −p $< > $*.C** **$(COMPILE.C) −o $% $*.C** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**

Table of Standard Implicit (Suffix) Rules		
Use	*Implicit Rule Name*	*Command Line*
FORTRAN 77	**.f**	**$(LINK.f) −o $@ $< $(LDLIBS)**
Files	**.f.o**	**$(COMPILE.f) $(OUTPUT_OPTION) $<**
	.f.a	**$(COMPILE.f) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.f	**$(LINK.f) −o $@ $< $(LDLIBS)**
	.f~	**$(GET) $(GFLAGS) −p $< > $*.f** **$(FC) $(FFLAGS) $(LDFLAGS) −o $@ $*.f**
	.f~.o	**$(GET) $(GFLAGS) −p $< > $*.f** **$(FC) $(FFLAGS) −c $*.f**
	.f~.a	**$(GET) $(GFLAGS) −p $< > $*.f** **$(COMPILE.f) −o $% $*.f** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.F	**$(LINK.F) −o $@ $< $(LDLIBS)**
	.F.o	**$(COMPILE.F) $(OUTPUT_OPTION) $<**
	.F.a	**$(COMPILE.F) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.F~	**$(GET) $(GFLAGS) −p $< > $*.F** **$(FC) $(FFLAGS) $(LDFLAGS) −o $@ $*.F**
	.F~.o	**$(GET) $(GFLAGS) −p $< > $*.F** **$(FC) $(FFLAGS) −c $*.F**
	.F~.a	**$(GET) $(GFLAGS) −p $< > $*.F** **$(COMPILE.F) −o $% $*.F** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**

Table of Standard Implicit (Suffix) Rules		
Use	*Implicit Rule Name*	*Command Line*
FORTRAN 90	**.f90**	**$(LINK.f90) −o $@ $< $(LDLIBS)**
Files	**.f90~**	**$(GET) $(GFLAGS) −p $< > $*.f90** **$(LINK.f90) −o $@ $*.f90 $(LDLIBS)**
	.f90.o	**$(COMPILE.f90) $(OUTPUT_OPTION) $<**
	.f90~.o	**$(GET) $(GFLAGS) −p $< > $*.f90** **$(COMPILE.f90) $(OUTPUT_OPTION) $*.f90**
	.f90.a	**$(COMPILE.f90) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.f90~.a	**$(GET) $(GFLAGS) −p $< > $*.f90** **$(COMPILE.f90) −o $% $*.f90** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.ftn	**$(LINK.ftn) −o $@ $< $(LDLIBS)**
	.ftn~	**$(GET) $(GFLAGS) −p $< > $*.ftn** **$(LINK.ftn) −o $@ $*.ftn $(LDLIBS)**
	.ftn.o	**$(COMPILE.ftn) $(OUTPUT_OPTION) $<**
	.ftn~.o	**$(GET) $(GFLAGS) −p $< > $*.ftn** **$(COMPILE.ftn) $(OUTPUT_OPTION) $*.ftn**
	.ftn.a	**$(COMPILE.ftn) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.ftn~.a	**$(GET) $(GFLAGS) −p $< > $*.ftn** **$(COMPILE.ftn) −o $% $*.ftn** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**

Table of Standard Implicit (Suffix) Rules		
Use	*Implicit Rule Name*	*Command Line*
lex Files	**.l**	**$(RM) $*.c** **$(LEX.l) $< > $*.c** **$(LINK.c) −o $@ $*.c $(LDLIBS)** **$(RM) $*.c**
	.l.c	**$(RM) $@** **$(LEX.l) $< > $@**
	.l.ln	**$(RM) $*.c** **$(LEX.l) $< > $*.c** **$(LINT.c) −o $@ −i $*.c** **$(RM) $*.c**
	.l.o	**$(RM) $*.c** **$(LEX.l) $< > $*.c** **$(COMPILE.c) −o $@ $*.c** **$(RM) $*.c**
	.l~	**$(GET) $(GFLAGS) −p $< > $*.l** **$(LEX) $(LFLAGS) $*.l** **$(CC) $(CFLAGS) −c lex.yy.c** **rm −f lex.yy.c** **mv lex.yy.c $@**
	.l~.c	**$(GET) $(GFLAGS) −p $< > $*.l** **$(LEX) $(LFLAGS) $*.l** **mv lex.yy.c $@**
	.l~.ln	**$(GET) $(GFLAGS) −p $< > $*.l** **$(RM) $*.c** **$(LEX.l) $*.l > $*.c** **$(LINT.c) −o $@ −i $*.c** **$(RM) $*.c**
	.l~.o	**$(GET) $(GFLAGS) −p $< > $*.l** **$(LEX) $(LFLAGS) $*.l** **$(CC) $(CFLAGS) −c lex.yy.c** **rm −f lex.yy.c** **mv lex.yy.c $@**

Table of Standard Implicit (Suffix) Rules		
Use	*Implicit Rule Name*	*Command Line*
Modula 2	**.mod**	**$(COMPILE.mod) −o $@ −e $@ $<**
Files	**.mod.o**	**$(COMPILE.mod) −o $@ $<**
	.def.sym	**$(COMPILE.def) −o $@ $<**
	.def~.sym	**$(GET) $(GFLAGS) −p $< > $*.def** **$(COMPILE.def) −o $@ $*.def**
	.mod~	**$(GET) $(GFLAGS) −p $< > $*.mod** **$(COMPILE.mod) −o $@ −e $@ $*.mod**
	.mod~.o	**$(GET) $(GFLAGS) −p $< > $*.mod** **$(COMPILE.mod) −o $@ $*.mod**
	.mod~.a	**$(GET) $(GFLAGS) −p $< > $*.mod** **$(COMPILE.mod) −o $% $*.mod** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
NeWS	**.cps.h**	**cps $*.cps**
	.cps~.h	**$(GET) $(GFLAGS) −p $< > $*.cps** **$(CPS) $(CPSFLAGS) $*.cps**
Pascal	**.p**	**$(LINK.p) −o $@ $< $(LDLIBS)**
Files	**.p.o**	**$(COMPILE.p) $(OUTPUT_OPTION) $<**
	.p~	**$(GET) $(GFLAGS) −p $< > $*.p** **$(LINK.p) −o $@ $*.p $(LDLIBS)**
	.p~.o	**$(GET) $(GFLAGS) −p $< > $*.p** **$(COMPILE.p) $(OUTPUT_OPTION) $*.p**
	.p~.a	**$(GET) $(GFLAGS) −p $< > $*.p** **$(COMPILE.p) −o $% $*.p** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
Ratfor	**.r**	**$(LINK.r) −o $@ $< $(LDLIBS)**
Files	**.r.o**	**$(COMPILE.r) $(OUTPUT_OPTION) $<**
	.r.a	**$(COMPILE.r) −o $% $<** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**
	.r~	**$(GET) $(GFLAGS) −p $< > $*.r** **$(LINK.r) −o $@ $*.r $(LDLIBS)**
	.r~.o	**$(GET) $(GFLAGS) −p $< > $*.r** **$(COMPILE.r) $(OUTPUT_OPTION) $*.r**
	.r~.a	**$(GET) $(GFLAGS) −p $< > $*.r** **$(COMPILE.r) −o $% $*.r** **$(AR) $(ARFLAGS) $@ $%** **$(RM) $%**

Table of Standard Implicit (Suffix) Rules

Use	*Implicit Rule Name*	*Command Line*
SCCS Files	**.SCCS_GET**	**sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@ −G$@**
	.SCCS_GET_POSIX	**sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@**
	.GET_POSIX	**$(GET) $(GFLAGS) s.$@**
Shell Scripts	**.sh**	**cat $< >$@** **chmod +x $@**
	.sh~	**$(GET) $(GFLAGS) −p $< > $*.sh** **cp $*.sh $@** **chmod a+x $@**
yacc Files	**.y**	**$(YACC.y) $<** **$(LINK.c) −o $@ y.tab.c $(LDLIBS)** **$(RM) y.tab.c**
	.y.c	**$(YACC.y) $<** **mv y.tab.c $@**
	.y.ln	**$(YACC.y) $<** **$(LINT.c) −o $@ −i y.tab.c** **$(RM) y.tab.c**
	.y.o	**$(YACC.y) $<** **$(COMPILE.c) −o $@ y.tab.c** **$(RM) y.tab.c**
	.y~	**$(GET) $(GFLAGS) −p $< > $*.y** **$(YACC) $(YFLAGS) $*.y** **$(COMPILE.c) −o $@ y.tab.c** **$(RM) y.tab.c**
	.y~.c	**$(GET) $(GFLAGS) −p $< > $*.y** **$(YACC) $(YFLAGS) $*.y** **mv y.tab.c $@**
	.y~.ln	**$(GET) $(GFLAGS) −p $< > $*.y** **$(YACC.y) $*.y** **$(LINT.c) −o $@ −i y.tab.c** **$(RM) y.tab.c**
	.y~.o	**$(GET) $(GFLAGS) −p $< > $*.y** **$(YACC) $(YFLAGS) $*.y** **$(CC) $(CFLAGS) −c y.tab.c** **rm −f y.tab.c** **mv y.tab.o $@**

make reads in the standard set of implicit rules from the file **/usr/share/lib/make/make.rules**, unless **−r** is in effect, or there is a **make.rules** file in the local directory that does not **include** that file.

The Suffixes List

The suffixes list is given as the list of dependencies for the '**.SUFFIXES:**' special-function target. The default list is contained in the **SUFFIXES** macro (See *Table of Predefined Macros* for the standard list of suffixes). You can define additional **.SUFFIXES:** targets; a **.SUFFIXES** target with no dependencies clears the list of suffixes. Order is significant within the list; **make** selects a rule that corresponds to the target's suffix and the first dependency-file suffix found in the list. To place suffixes at the head of the list, clear the list and replace it with the new suffixes, followed by the default list:

.SUFFIXES:
.SUFFIXES: *suffixes* **$(SUFFIXES)**

A tilde (˜) indicates that if a dependency file with the indicated suffix (minus the ˜) is under SCCS its most recent version should be retrieved, if necessary, before the target is processed.

Library Maintenance

A target name of the form:

lib(*member* ...)

refers to a member, or a space-separated list of members, in an **ar**(1) library.

The dependency of the library member on the corresponding file must be given as an explicit entry in the makefile. This can be handled by a pattern matching rule of the form:

lib(%.*s*)**:** %.*s*

where .*s* is the suffix of the member; this suffix is typically **.o** for object libraries.

A target name of the form

lib((*symbol*))

refers to the member of a randomized object library that defines the entry point named *symbol*.

Command Execution

Command lines are executed one at a time, *each by its own process or shell.* Shell commands, notably **cd**, are ineffectual across an unescaped NEWLINE in the makefile. A line is printed (after macro expansion) just before being executed. This is suppressed if it starts with a '@', if there is a '**.SILENT:**' entry in the makefile, or if **make** is run with the −**s** option. Although the −**n** option specifies printing without execution, lines containing the macro **$(MAKE)** are executed regardless, and lines containing the @ special character are printed. The −**t** (touch) option updates the modification date of a file without executing any rules. This can be dangerous when sources are maintained by more than one person.

make invokes the shell with the −**e** (exit-on-errors) argument. Thus, with semicolon-separated command sequences, execution of the later commands depends on the success of the former. This behavior can be overridden by starting the command line with a '**-**', or by writing a shell script that returns a non-zero status only as it finds appropriate.

Bourne Shell Constructs

To use the Bourne shell **if** control structure for branching, use a command line of the form:

```
if expression ; \
then command ; \
        ... ; \
else command ; \
        ... ; \
fi
```

Although composed of several input lines, the escaped NEWLINE characters insure that **make** treats them all as one (shell) command line.

To use the Bourne shell **for** control structure for loops, use a command line of the form:

```
for var in list ; \
        do command; \
        . . . ; \
done
```

To refer to a shell variable, use a double-dollar-sign (**$$**). This prevents expansion of the dollar-sign by **make**.

Command Substitutions

To incorporate the standard output of a shell command in a macro, use a definition of the form:

MACRO **:sh =***command*

The command is executed only once, standard error output is discarded, and NEWLINE characters are replaced with SPACEs. If the command has a non-zero exit status, **make** halts with an error.

To capture the output of a shell command in a macro reference, use a reference of the form:

$(*MACRO* **:sh)**

where *MACRO* is the name of a macro containing a valid Bourne shell command line. In this case, the command is executed whenever the reference is evaluated. As with shell command substitutions, the reference is replaced with the standard output of the command. If the command has a non-zero exit status, **make** halts with an error.

Signals

INT, **SIGTERM**, and **QUIT** signals received from the keyboard halt **make** and remove the target file being processed unless that target is in the dependency list for **.PRECIOUS:**.

EXAMPLES

This 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**) along with a common file **incl.h**:

```
pgm: a.o b.o
        $(LINK.c) –o $@ a.o b.o
a.o: incl.h a.c
        cc –c a.c
b.o: incl.h b.c
        cc –c b.c
```

The following makefile uses implicit rules to express the same dependencies:

```
pgm: a.o b.o
        cc a.o b.o –o pgm
a.o b.o: incl.h
```

ENVIRONMENT

See **environ**(5) for descriptions of the following environment variables that affect the execution of **make**: **LC_CTYPE**, **LC_MESSAGES**, and **NLSPATH**.

KEEP_STATE
: This environment variable has the same effect as the **.KEEP_STATE:** special-function target. It enables command dependencies, hidden dependencies and writing of the state file.

USE_SVR4_MAKE
: This environment variable causes **make** to invoke the generic System V version of **make** (**/usr/ccs/lib/svr4.make**). See **sysV-make**(1).

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

1. The characters are option letters without the leading hyphens or blank character separation used on a command line.
2. The characters are formatted in a manner similar to a portion of the **make** command line: options are preceded by hyphens and blank-character-separated. The *macro=name* macro definition operands can also be included. The difference between the contents of **MAKEFLAGS** and the command line is that the contents of the variable will not be subjected to the word expansions (see **wordexp**(3C)) associated with parsing the command line values.

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

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

EXIT STATUS

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

0 successful completion

>0 an error occurred

FILES

makefile	
Makefile	current version(s) of **make** description file
s.makefile	
s.Makefile	SCCS history files for the above makefile(s) in the current directory
SCCS/s.makefile	
SCCS/s.Makefile	SCCS history files for the above makefile(s)
make.rules	default file for user-defined targets, macros, and implicit rules
/usr/share/lib/make/make.rules	makefile for standard implicit rules and macros (not read if **make.rules** is)
.make.state	state file in the local directory

SEE ALSO

ar(1), **cd**(1), **lex**(1), **sh**(1), **sccs-get**(1), **yacc**(1), **passwd**(4)

Solaris Advanced User's Guide
Programming Utilities Guide

DIAGNOSTICS

Don't know how to make target '*target***'**
There is no makefile entry for *target*, and none of **make**'s implicit rules apply (there is no dependency file with a suffix in the suffixes list, or the target's suffix is not in the list).

*** *target* **removed.**
make was interrupted while building *target*. Rather than leaving a partially-completed version that is newer than its dependencies, **make** removes the file named *target*.

*** *target* **not removed.**
make was interrupted while building *target* and *target* was not present in the directory.

*** *target* **could not be removed,** *reason*
make was interrupted while building *target*, which was not removed for the indicated reason.

Read of include file '*file***' failed**
The makefile indicated in an **include** directive was not found, or was inaccessible.

Loop detected when expanding macro value '*macro***'**
A reference to the macro being defined was found in the definition.

Could not write state file '*file***'**
You used the **.KEEP_STATE:** target, but do not have write permission on the state file.

*** **Error code** *n*
The previous shell command returned a nonzero error code.

**** signal message*
The previous shell command was aborted due to a signal. If '– **core dumped**' appears after the message, a **core** file was created.

Conditional macro conflict encountered
Displayed only when –**d** is in effect, this message indicates that two or more parallel targets currently being processed depend on a target which is built differently for each by virtue of conditional macros. Since the target cannot simultaneously satisfy both dependency relationships, it is conflicted.

BUGS

Some commands return nonzero status inappropriately; to overcome this difficulty, prefix the offending command line in the rule with a '–'.

Filenames with the characters '**=**', '**:**', or '**@**', do not work.

You cannot build **file.o** from **lib(file.o)**.

Options supplied by **MAKEFLAGS** should be reported for nested **make** commands. Use the –**d** option to find out what options the nested command picks up from **MAKEFLAGS**.

This version of **make** is incompatible in certain respects with previous versions:

- The –**d** option output is much briefer in this version. –**dd** now produces the equivalent voluminous output.
- **make** attempts to derive values for the dynamic macros '**$***', '**$<**', and '**$?**', while processing explicit targets. It uses the same method as for implicit rules; in some cases this can lead either to unexpected values, or to an empty value being assigned. (Actually, this was true for earlier versions as well, even though the documentation stated otherwise.)
- **make** no longer searches for SCCS history "(s.)" files.
- Suffix replacement in macro references are now applied after the macro is expanded.

There is no guarantee that makefiles created for this version of **make** will work with earlier versions.

If there is no **make.rules** file in the current directory, and the file **/usr/share/lib/make/make.rules** is missing, **make** stops before processing any targets. To force **make** to run anyway, create an empty **make.rules** file in the current directory.

Once a dependency is made, **make** assumes the dependency file is present for the remainder of the run. If a rule subsequently removes that file and future targets depend on its existence, unexpected errors may result.

When hidden dependency checking is in effect, the **$?** macro's value includes the names of hidden dependencies. This can lead to improper filename arguments to commands when **$?** is used in a rule.

Pattern replacement macro references cannot be used in the dependency list of a pattern matching rule.

Unlike previous versions, this version of **make** strips a leading '**./**' from the value of the '**$@**' dynamic macro.

With automatic SCCS retrieval, this version of **make** does not support tilde suffix rules.

The only dynamic macro whose value is strictly determined when used in a dependency list is **$@** (takes the form '**$$@**').

make invokes the shell with the −**e** argument. This cannot be inferred from the syntax of the rule alone.