Go to the first, previous, next, last section, table of contents.

Using Implicit Rules

Certain standard ways of remaking target files are used very often. For example, one customary way to make an object file is from a C source file using the C compiler, `cc`.

Implicit rules tell `make` how to use customary techniques so that you do not have to specify them in detail when you want to use them. For example, there is an implicit rule for C compilation. File names determine which implicit rules are run. For example, C compilation typically takes a `` `.c' `` file and makes a `` `.o' `` file. So `make` applies the implicit rule for C compilation when it sees this combination of file name endings.

A chain of implicit rules can apply in sequence; for example, `make` will remake a `` `.o' `` file from a `` `.y' `` file by way of a `` `.c' `` file. See section Chains of Implicit Rules.

The built-in implicit rules use several variables in their commands so that, by changing the values of the variables, you can change the way the implicit rule works. For example, the variable `CFLAGS` controls the flags given to the C compiler by the implicit rule for C compilation. See section Variables Used by Implicit Rules.

You can define your own implicit rules by writing **pattern rules**. See section Defining and Redefining Pattern Rules.

Suffix rules are a more limited way to define implicit rules. Pattern rules are more general and clearer, but suffix rules are retained for compatibility. See section Old-Fashioned Suffix Rules.

Using Implicit Rules

To allow `make` to find a customary method for updating a target file, all you have to do is refrain from specifying commands yourself. Either write a rule with no command lines, or don't write a rule at all. Then `make` will figure out which implicit rule to use based on which kind of source file exists or can be made.

For example, suppose the makefile looks like this:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Because you mention `` `foo.o' `` but do not give a rule for it, `make` will automatically look for an implicit rule that tells how to update it. This happens whether or not the file `` `foo.o' `` currently exists.

If an implicit rule is found, it can supply both commands and one or more prerequisites (the source files). You would want to write a rule for `` `foo.o' `` with no command lines if you need to specify additional prerequisites, such as header files, that the implicit rule cannot supply.

Each implicit rule has a target pattern and prerequisite patterns. There may be many implicit rules with the same target pattern. For example, numerous rules make `` `.o' `` files: one, from a `` `.c' `` file with the C compiler; another, from a `` `.p' `` file with the Pascal compiler; and so on. The rule that actually applies is the one whose prerequisites exist or can be made. So, if you have a file `` `foo.c' ``, `make` will run the C compiler; otherwise, if you have a file `` `foo.p' ``, `make` will run the Pascal compiler; and so on.

Of course, when you write the makefile, you know which implicit rule you want `make` to use, and you know it will choose that one because you know which possible prerequisite files are supposed to exist. See section Catalogue of Implicit Rules, for a catalogue of all the predefined implicit rules.

Above, we said an implicit rule applies if the required prerequisites "exist or can be made". A file "can be made" if it is mentioned explicitly in the makefile as a target or a prerequisite, or if an implicit rule can be recursively found for how to make it. When an implicit prerequisite is the result of another implicit rule, we say that **chaining** is occurring. See section Chains of Implicit Rules.

In general, `make` searches for an implicit rule for each target, and for each double-colon rule, that has no commands. A file that is mentioned only as a prerequisite is considered a target whose rule specifies nothing, so implicit rule search happens for it. See section Implicit Rule Search Algorithm, for the details of how the search is done.

Note that explicit prerequisites do not influence implicit rule search. For example, consider this explicit rule:

```
foo.o: foo.p
```

The prerequisite on `` `foo.p' `` does not necessarily mean that `make` will remake `` `foo.o' `` according to the implicit rule to make an object file, a `` `.o' `` file, from a Pascal source file, a `` `.p' `` file. For example, if `` `foo.c' `` also exists, the implicit rule to make an object file from a C source file is used instead, because it appears before the Pascal rule in the list of predefined implicit rules (see section Catalogue of Implicit Rules).

If you do not want an implicit rule to be used for a target that has no commands, you can give that target empty commands by writing a semicolon (see section Using Empty Commands).

Catalogue of Implicit Rules

Here is a catalogue of predefined implicit rules which are always available unless the makefile explicitly overrides or cancels them. See section Canceling Implicit Rules, for information on canceling or overriding an implicit rule. The `-r' or `--no-builtin-rules' option cancels all predefined rules.

Not all of these rules will always be defined, even when the `-r' option is not given. Many of the predefined implicit rules are implemented in `make` as suffix rules, so which ones will be defined depends on the **suffix list** (the list of prerequisites of the special target `.SUFFIXES`). The default suffix list is: `.out`, `.a`, `.ln`, `.o`, `.c`, `.cc`, `.C`, `.p`, `.f`, `.F`, `.r`, `.y`, `.l`, `.s`, `.S`, `.mod`, `.sym`, `.def`, `.h`, `.info`, `.dvi`, `.tex`, `.texinfo`, `.texi`, `.txinfo`, `.w`, `.ch` `.web`, `.sh`, `.elc`, `.el`. All of the implicit rules described below whose prerequisites have one of these suffixes are actually suffix rules. If you modify the suffix list, the only predefined suffix rules in effect will be those named by one or two of the suffixes that are on the list you specify; rules whose suffixes fail to be on the list are disabled. See section Old-Fashioned Suffix Rules, for full details on suffix rules.

Compiling C programs
: `*n*.o' is made automatically from `*n*.c' with a command of the form `$(CC) -c $(CPPFLAGS) $(CFLAGS)'.

Compiling C++ programs
: `*n*.o' is made automatically from `*n*.cc' or `*n*.C' with a command of the form `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)'. We encourage you to use the suffix `.cc' for C++ source files instead of `.C'.

Compiling Pascal programs
: `*n*.o' is made automatically from `*n*.p' with the command `$(PC) -c $(PFLAGS)'.

Compiling Fortran and Ratfor programs
: `*n*.o' is made automatically from `*n*.r', `*n*.F' or `*n*.f' by running the Fortran compiler. The precise command used is as follows:

  `.f'
  : `$(FC) -c $(FFLAGS)'.

  `.F'
  : `$(FC) -c $(FFLAGS) $(CPPFLAGS)'.

  `.r'
  : `$(FC) -c $(FFLAGS) $(RFLAGS)'.

Preprocessing Fortran and Ratfor programs
: `*n*.f' is made automatically from `*n*.r' or `*n*.F'. This rule runs just the preprocessor to convert a Ratfor or preprocessable Fortran program into a strict Fortran program. The precise command used is as follows:

  `.F'
  : `$(FC) -F $(CPPFLAGS) $(FFLAGS)'.

  `.r'
  : `$(FC) -F $(FFLAGS) $(RFLAGS)'.

Compiling Modula-2 programs

`` `n.sym' `` is made from `` `n.def' `` with a command of the form `` `$(M2C) $(M2FLAGS) $(DEFFLAGS)' ``. `` `n.o' `` is made from `` `n.mod' ``; the form is: `` `$(M2C) $(M2FLAGS) $(MODFLAGS)' ``.

Assembling and preprocessing assembler programs

`` `n.o' `` is made automatically from `` `n.s' `` by running the assembler, `as`. The precise command is `` `$(AS) $(ASFLAGS)' ``. `` `n.s' `` is made automatically from `` `n.S' `` by running the C preprocessor, `cpp`. The precise command is `` `$(CPP) $(CPPFLAGS)' ``.

Linking a single object file

`` `n' `` is made automatically from `` `n.o' `` by running the linker (usually called `ld`) via the C compiler. The precise command used is `` `$(CC) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)' ``. This rule does the right thing for a simple program with only one source file. It will also do the right thing if there are multiple object files (presumably coming from various other source files), one of which has a name matching that of the executable file. Thus,

```
x: y.o z.o
```

when `` `x.c' ``, `` `y.c' `` and `` `z.c' `` all exist will execute:

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

In more complicated cases, such as when there is no object file whose name derives from the executable file name, you must write an explicit command for linking. Each kind of file automatically made into `` `.o' `` object files will be automatically linked by using the compiler (`` `$(CC)' ``, `` `$(FC)' `` or `` `$(PC)' ``; the C compiler `` `$(CC)' `` is used to assemble `` `.s' `` files) without the `` `-c' `` option. This could be done by using the `` `.o' `` object files as intermediates, but it is faster to do the compiling and linking in one step, so that's how it's done.

Yacc for C programs

`` `n.c' `` is made automatically from `` `n.y' `` by running Yacc with the command `` `$(YACC) $(YFLAGS)' ``.

Lex for C programs

`` `n.c' `` is made automatically from `` `n.l' `` by by running Lex. The actual command is `` `$(LEX) $(LFLAGS)' ``.

Lex for Ratfor programs

`` `n.r' `` is made automatically from `` `n.l' `` by by running Lex. The actual command is `` `$(LEX) $(LFLAGS)' ``. The convention of using the same suffix `` `.l' `` for all Lex files regardless of whether they produce C code or Ratfor code makes it impossible for `make` to determine automatically which of the two languages you are using in any particular case. If `make` is called upon to remake an object file from a `` `.l' `` file, it must guess which compiler to use. It will guess the C compiler, because that is more common. If you are using

Ratfor, make sure `make` knows this by mentioning `` `n.r' `` in the makefile. Or, if you are using Ratfor exclusively, with no C files, remove `` `.c' `` from the list of implicit rule suffixes with:

```
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
```

Making Lint Libraries from C, Yacc, or Lex programs
: `` `n.ln' `` is made from `` `n.c' `` by running `lint`. The precise command is `` `$(LINT) $(LINTFLAGS) $(CPPFLAGS) -i' ``. The same command is used on the C code produced from `` `n.y' `` or `` `n.l' ``.

TeX and Web
: `` `n.dvi' `` is made from `` `n.tex' `` with the command `` `$(TEX)' ``. `` `n.tex' `` is made from `` `n.web' `` with `` `$(WEAVE)' ``, or from `` `n.w' `` (and from `` `n.ch' `` if it exists or can be made) with `` `$(CWEAVE)' ``. `` `n.p' `` is made from `` `n.web' `` with `` `$(TANGLE)' `` and `` `n.c' `` is made from `` `n.w' `` (and from `` `n.ch' `` if it exists or can be made) with `` `$(CTANGLE)' ``.

Texinfo and Info
: `` `n.dvi' `` is made from `` `n.texinfo' ``, `` `n.texi' ``, or `` `n.txinfo' ``, with the command `` `$(TEXI2DVI) $(TEXI2DVI_FLAGS)' ``. `` `n.info' `` is made from `` `n.texinfo' ``, `` `n.texi' ``, or `` `n.txinfo' ``, with the command `` `$(MAKEINFO) $(MAKEINFO_FLAGS)' ``.

RCS
: Any file `` `n' `` is extracted if necessary from an RCS file named either `` `n,v' `` or `` `RCS/n,v' ``. The precise command used is `` `$(CO) $(COFLAGS)' ``. `` `n' `` will not be extracted from RCS if it already exists, even if the RCS file is newer. The rules for RCS are terminal (see section Match-Anything Pattern Rules), so RCS files cannot be generated from another source; they must actually exist.

SCCS
: Any file `` `n' `` is extracted if necessary from an SCCS file named either `` `s.n' `` or `` `SCCS/s.n' ``. The precise command used is `` `$(GET) $(GFLAGS)' ``. The rules for SCCS are terminal (see section Match-Anything Pattern Rules), so SCCS files cannot be generated from another source; they must actually exist. For the benefit of SCCS, a file `` `n' `` is copied from `` `n.sh' `` and made executable (by everyone). This is for shell scripts that are checked into SCCS. Since RCS preserves the execution permission of a file, you do not need to use this feature with RCS. We recommend that you avoid using of SCCS. RCS is widely held to be superior, and is also free. By choosing free software in place of comparable (or inferior) proprietary software, you support the free software movement.

Usually, you want to change only the variables listed in the table above, which are documented in the following section.

However, the commands in built-in implicit rules actually use variables such as `COMPILE.c`, `LINK.p`, and `PREPROCESS.S`, whose values contain the commands listed above.

`make` follows the convention that the rule to compile a `` `.x' `` source file uses the variable `COMPILE.x`. Similarly, the rule to produce an executable from a `` `.x' `` file uses `LINK.x`; and the rule to preprocess a `` `.x' `` file uses `PREPROCESS.x`.

Every rule that produces an object file uses the variable `OUTPUT_OPTION`. `make` defines this variable either to contain `` `-o $@' ``, or to be empty, depending on a compile-time option. You need the `` `-o' `` option to ensure that the output goes into the right file when the source file is in a different directory, as when using `VPATH` (see section Searching Directories for Prerequisites). However, compilers on some systems do not accept a `` `-o' `` switch for object files. If you use such a system, and use `VPATH`, some compilations will put their output in the wrong place. A possible workaround for this problem is to give `OUTPUT_OPTION` the value `` `; mv $*.o $@' ``.

Variables Used by Implicit Rules

The commands in built-in implicit rules make liberal use of certain predefined variables. You can alter these variables in the makefile, with arguments to `make`, or in the environment to alter how the implicit rules work without redefining the rules themselves. You can cancel all variables used by implicit rules with the `` `-R' `` or `` `--no-builtin-variables' `` option.

For example, the command used to compile a C source file actually says `` `$(CC) -c $(CFLAGS) $(CPPFLAGS)' ``. The default values of the variables used are `` `cc' `` and nothing, resulting in the command `` `cc -c' ``. By redefining `` `CC' `` to `` `ncc' ``, you could cause `` `ncc' `` to be used for all C compilations performed by the implicit rule. By redefining `` `CFLAGS' `` to be `` `-g' ``, you could pass the `` `-g' `` option to each compilation. *All* implicit rules that do C compilation use `` `$(CC)' `` to get the program name for the compiler and *all* include `` `$(CFLAGS)' `` among the arguments given to the compiler.

The variables used in implicit rules fall into two classes: those that are names of programs (like `CC`) and those that contain arguments for the programs (like `CFLAGS`). (The "name of a program" may also contain some command arguments, but it must start with an actual executable program name.) If a variable value contains more than one argument, separate them with spaces.

Here is a table of variables used as names of programs in built-in rules:

`AR`	Archive-maintaining program; default `` `ar' ``.
`AS`	Program for doing assembly; default `` `as' ``.
`CC`	Program for compiling C programs; default `` `cc' ``.
`CXX`	Program for compiling C++ programs; default `` `g++' ``.

`CO`
: Program for extracting a file from RCS; default `` `co' ``.

`CPP`
: Program for running the C preprocessor, with results to standard output; default `` `$(CC) -E' ``.

`FC`
: Program for compiling or preprocessing Fortran and Ratfor programs; default `` `f77' ``.

`GET`
: Program for extracting a file from SCCS; default `` `get' ``.

`LEX`
: Program to use to turn Lex grammars into C programs or Ratfor programs; default `` `lex' ``.

`PC`
: Program for compiling Pascal programs; default `` `pc' ``.

`YACC`
: Program to use to turn Yacc grammars into C programs; default `` `yacc' ``.

`YACCR`
: Program to use to turn Yacc grammars into Ratfor programs; default `` `yacc -r' ``.

`MAKEINFO`
: Program to convert a Texinfo source file into an Info file; default `` `makeinfo' ``.

`TEX`
: Program to make TeX DVI files from TeX source; default `` `tex' ``.

`TEXI2DVI`
: Program to make TeX DVI files from Texinfo source; default `` `texi2dvi' ``.

`WEAVE`
: Program to translate Web into TeX; default `` `weave' ``.

`CWEAVE`
: Program to translate C Web into TeX; default `` `cweave' ``.

`TANGLE`
: Program to translate Web into Pascal; default `` `tangle' ``.

`CTANGLE`
: Program to translate C Web into C; default `` `ctangle' ``.

`RM`
: Command to remove a file; default `` `rm -f' ``.

Here is a table of variables whose values are additional arguments for the programs above. The default values for all of these is the empty string, unless otherwise noted.

`ARFLAGS`
: Flags to give the archive-maintaining program; default `` `rv' ``.

`ASFLAGS`
: Extra flags to give to the assembler (when explicitly invoked on a `` `.s' `` or `` `.S' `` file).

`CFLAGS`

Extra flags to give to the C compiler.

`CXXFLAGS`
: Extra flags to give to the C++ compiler.

`COFLAGS`
: Extra flags to give to the RCS `co` program.

`CPPFLAGS`
: Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers).

`FFLAGS`
: Extra flags to give to the Fortran compiler.

`GFLAGS`
: Extra flags to give to the SCCS `get` program.

`LDFLAGS`
: Extra flags to give to compilers when they are supposed to invoke the linker, `` `ld' ``.

`LFLAGS`
: Extra flags to give to Lex.

`PFLAGS`
: Extra flags to give to the Pascal compiler.

`RFLAGS`
: Extra flags to give to the Fortran compiler for Ratfor programs.

`YFLAGS`
: Extra flags to give to Yacc.

Chains of Implicit Rules

Sometimes a file can be made by a sequence of implicit rules. For example, a file `` `n.o' `` could be made from `` `n.y' `` by running first Yacc and then `cc`. Such a sequence is called a **chain**.

If the file `` `n.c' `` exists, or is mentioned in the makefile, no special searching is required: `make` finds that the object file can be made by C compilation from `` `n.c' ``; later on, when considering how to make `` `n.c' ``, the rule for running Yacc is used. Ultimately both `` `n.c' `` and `` `n.o' `` are updated.

However, even if `` `n.c' `` does not exist and is not mentioned, `make` knows how to envision it as the missing link between `` `n.o' `` and `` `n.y' ``! In this case, `` `n.c' `` is called an **intermediate file**. Once `make` has decided to use the intermediate file, it is entered in the data base as if it had been mentioned in the makefile, along with the implicit rule that says how to create it.

Intermediate files are remade using their rules just like all other files. But intermediate files are treated differently in two ways.

The first difference is what happens if the intermediate file does not exist. If an ordinary file *b* does not exist, and `make` considers a target that depends on *b*, it invariably creates *b* and then updates the target from *b*. But if *b* is an intermediate file, then `make` can leave well enough alone. It won't bother updating *b*, or the ultimate target, unless some prerequisite of *b* is newer than that target or there is some other reason to update that target.

The second difference is that if `make` *does* create *b* in order to update something else, it deletes *b* later on after it is no longer needed. Therefore, an intermediate file which did not exist before `make` also does not exist after `make`. `make` reports the deletion to you by printing a ``rm -f'` command showing which file it is deleting.

Ordinarily, a file cannot be intermediate if it is mentioned in the makefile as a target or prerequisite. However, you can explicitly mark a file as intermediate by listing it as a prerequisite of the special target `.INTERMEDIATE`. This takes effect even if the file is mentioned explicitly in some other way.

You can prevent automatic deletion of an intermediate file by marking it as a **secondary** file. To do this, list it as a prerequisite of the special target `.SECONDARY`. When a file is secondary, `make` will not create the file merely because it does not already exist, but `make` does not automatically delete the file. Marking a file as secondary also marks it as intermediate.

You can list the target pattern of an implicit rule (such as ``%.o'`) as a prerequisite of the special target `.PRECIOUS` to preserve intermediate files made by implicit rules whose target patterns match that file's name; see section Interrupting or Killing `make`.

A chain can involve more than two implicit rules. For example, it is possible to make a file ``foo'` from ``RCS/foo.y,v'` by running RCS, Yacc and `cc`. Then both ``foo.y'` and ``foo.c'` are intermediate files that are deleted at the end.

No single implicit rule can appear more than once in a chain. This means that `make` will not even consider such a ridiculous thing as making ``foo'` from ``foo.o.o'` by running the linker twice. This constraint has the added benefit of preventing any infinite loop in the search for an implicit rule chain.

There are some special implicit rules to optimize certain cases that would otherwise be handled by rule chains. For example, making ``foo'` from ``foo.c'` could be handled by compiling and linking with separate chained rules, using ``foo.o'` as an intermediate file. But what actually happens is that a special rule for this case does the compilation and linking with a single `cc` command. The optimized rule is used in preference to the step-by-step chain because it comes earlier in the ordering of rules.

Defining and Redefining Pattern Rules

You define an implicit rule by writing a **pattern rule**. A pattern rule looks like an ordinary rule, except that its target contains the character ``%'` (exactly one of them). The target is considered a pattern for matching file names; the `` `%' `` can match any nonempty substring, while other characters match only themselves. The prerequisites likewise use `` `%' `` to show how their names relate to the target name.

Thus, a pattern rule `` `%.o : %.c' `` says how to make any file `` `*stem*.o' `` from another file `` `*stem*.c' ``.

Note that expansion using `` `%' `` in pattern rules occurs **after** any variable or function expansions, which take place when the makefile is read. See section How to Use Variables, and section Functions for Transforming Text.

Introduction to Pattern Rules

A pattern rule contains the character `` `%' `` (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the `` `%' `` matches any nonempty substring, while other characters match only themselves.

For example, `` `%.c' `` as a pattern matches any file name that ends in `` `.c' ``. `` `s.%.c' `` as a pattern matches any file name that starts with `` `s.' ``, ends in `` `.c' `` and is at least five characters long. (There must be at least one character to match the `` `%' ``.) The substring that the `` `%' `` matches is called the **stem**.

`` `%' `` in a prerequisite of a pattern rule stands for the same stem that was matched by the `` `%' `` in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration, and its prerequisite patterns must name files that exist or can be made. These files become prerequisites of the target.

Thus, a rule of the form

```
%.o : %.c ; command...
```

specifies how to make a file `` `*n*.o' ``, with another file `` `*n*.c' `` as its prerequisite, provided that `` `*n*.c' `` exists or can be made.

There may also be prerequisites that do not use `` `%' ``; such a prerequisite attaches to every file made by this pattern rule. These unvarying prerequisites are useful occasionally.

A pattern rule need not have any prerequisites that contain `` `%' ``, or in fact any prerequisites at all. Such a rule is effectively a general wildcard. It provides a way to make any file that matches the target pattern. See section Defining Last-Resort Default Rules.

Pattern rules may have more than one target. Unlike normal rules, this does not act as many different rules with the same prerequisites and commands. If a pattern rule has multiple targets, `make` knows that the rule's commands are responsible for making all of the targets. The commands are executed only once to make all the targets. When searching for a pattern rule to match a target, the target patterns of a rule other than the one that matches the target in need of a rule are incidental: `make` worries only about giving commands and prerequisites to the file presently in question. However, when this file's commands are run, the other targets are marked as having been updated themselves.

The order in which pattern rules appear in the makefile is important since this is the order in which they are considered. Of equally applicable rules, only the first one found is used. The rules you write take precedence over those that are built in. Note however, that a rule whose prerequisites actually exist or are mentioned always takes priority over a rule with prerequisites that must be made by chaining other implicit rules.

Pattern Rule Examples

Here are some examples of pattern rules actually predefined in `make`. First, the rule that compiles \``.c`' files into \``.o`' files:

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

defines a rule that can make any file \``x.o`' from \``x.c`'. The command uses the automatic variables \``$@`' and \``$<`' to substitute the names of the target file and the source file in each case where the rule applies (see section Automatic Variables).

Here is a second built-in rule:

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

defines a rule that can make any file \``x`' whatsoever from a corresponding file \``x,v`' in the subdirectory \``RCS`'. Since the target is \``%`', this rule will apply to any file whatever, provided the appropriate prerequisite file exists. The double colon makes the rule **terminal**, which means that its prerequisite may not be an intermediate file (see section Match-Anything Pattern Rules).

This pattern rule has two targets:

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

This tells `make` that the command \``bison -d x.y`' will make both \``x.tab.c`' and \``x.tab.h`'. If the file \``foo`' depends on the files \``parse.tab.o`' and \``scan.o`' and the file \``scan.o`' depends on the file \``parse.tab.h`', when \``parse.y`' is changed, the command \``bison -d parse.y`' will be executed only once, and the prerequisites of both \``parse.tab.o`' and \``scan.o`' will be satisfied. (Presumably the file \``parse.tab.o`' will be recompiled from `` `parse.tab.c' `` and the file `` `scan.o' `` from `` `scan.c' ``, while `` `foo' `` is linked from `` `parse.tab.o' ``, `` `scan.o' ``, and its other prerequisites, and it will execute happily ever after.)

Automatic Variables

Suppose you are writing a pattern rule to compile a `` `.c' `` file into a `` `.o' `` file: how do you write the `` `cc' `` command so that it operates on the right source file name? You cannot write the name in the command, because the name is different each time the implicit rule is applied.

What you do is use a special feature of `make`, the **automatic variables**. These variables have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule. In this example, you would use `` `$@' `` for the object file name and `` `$<' `` for the source file name.

Here is a table of automatic variables:

`$@`
: The file name of the target of the rule. If the target is an archive member, then `` `$@' `` is the name of the archive file. In a pattern rule that has multiple targets (see section Introduction to Pattern Rules), `` `$@' `` is the name of whichever target caused the rule's commands to be run.

`$%`
: The target member name, when the target is an archive member. See section Using `make` to Update Archive Files. For example, if the target is `` `foo.a(bar.o)' `` then `` `$%' `` is `` `bar.o' `` and `` `$@' `` is `` `foo.a' ``. `` `$%' `` is empty when the target is not an archive member.

`$<`
: The name of the first prerequisite. If the target got its commands from an implicit rule, this will be the first prerequisite added by the implicit rule (see section Using Implicit Rules).

`$?`
: The names of all the prerequisites that are newer than the target, with spaces between them. For prerequisites which are archive members, only the member named is used (see section Using `make` to Update Archive Files).

`$^`
: The names of all the prerequisites, with spaces between them. For prerequisites which are archive members, only the member named is used (see section Using `make` to Update Archive Files). A target has only one prerequisite on each other file it depends on, no matter how many times each file is listed as a prerequisite. So if you list a prerequisite more than once for a target, the value of `$^` contains just one copy of the name.

`$+`
: This is like `` `$^' ``, but prerequisites listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use

in linking commands where it is meaningful to repeat library file names in a particular order.

`$*`
: The stem with which an implicit rule matches (see section How Patterns Match). If the target is `` `dir/a.foo.b' `` and the target pattern is `` `a.%.b' `` then the stem is `` `dir/foo' ``. The stem is useful for constructing names of related files. In a static pattern rule, the stem is part of the file name that matched the `` `%' `` in the target pattern. In an explicit rule, there is no stem; so `` `$*' `` cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see section Old-Fashioned Suffix Rules), `` `$*' `` is set to the target name minus the suffix. For example, if the target name is `` `foo.c' ``, then `` `$*' `` is set to `` `foo' ``, since `` `.c' `` is a suffix. GNU `make` does this bizarre thing only for compatibility with other implementations of `make`. You should generally avoid using `` `$*' `` except in implicit rules or static pattern rules. If the target name in an explicit rule does not end with a recognized suffix, `` `$*' `` is set to the empty string for that rule.

`` `$?' `` is useful even in explicit rules when you wish to operate on only the prerequisites that have changed. For example, suppose that an archive named `` `lib' `` is supposed to contain copies of several object files. This rule copies just the changed object files into the archive:

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

Of the variables listed above, four have values that are single file names, and three have values that are lists of file names. These seven have variants that get just the file's directory name or just the file name within the directory. The variant variables' names are formed by appending `` `D' `` or `` `F' ``, respectively. These variants are semi-obsolete in GNU `make` since the functions `dir` and `notdir` can be used to get a similar effect (see section Functions for File Names). Note, however, that the `` `F' `` variants all omit the trailing slash which always appears in the output of the `dir` function. Here is a table of the variants:

`` `$(@D)' ``
: The directory part of the file name of the target, with the trailing slash removed. If the value of `` `$@' `` is `` `dir/foo.o' `` then `` `$(@D)' `` is `` `dir' ``. This value is `` `.' `` if `` `$@' `` does not contain a slash.

`` `$(@F)' ``
: The file-within-directory part of the file name of the target. If the value of `` `$@' `` is `` `dir/foo.o' `` then `` `$(@F)' `` is `` `foo.o' ``. `` `$(@F)' `` is equivalent to `` `$(notdir $@)' ``.

`` `$(*D)' ``
`` `$(*F)' ``
: The directory part and the file-within-directory part of the stem; `` `dir' `` and `` `foo' `` in this example.

`` `$(%D)' ``
`` `$(%F)' ``
: The directory part and the file-within-directory part of the target archive

member name. This makes sense only for archive member targets of the form `` `archive(member)' `` and is useful only when *member* may contain a directory name. (See section Archive Members as Targets.)

`` `$(<D)' ``
`` `$(<F)' ``
: The directory part and the file-within-directory part of the first prerequisite.

`` `$(^D)' ``
`` `$(^F)' ``
: Lists of the directory parts and the file-within-directory parts of all prerequisites.

`` `$(?D)' ``
`` `$(?F)' ``
: Lists of the directory parts and the file-within-directory parts of all prerequisites that are newer than the target.

Note that we use a special stylistic convention when we talk about these automatic variables; we write "the value of `` `$<' ``", rather than "the variable `<`" as we would write for ordinary variables such as `objects` and `CFLAGS`. We think this convention looks more natural in this special case. Please do not assume it has a deep significance; `` `$<' `` refers to the variable named `<` just as `` `$(CFLAGS)' `` refers to the variable named `CFLAGS`. You could just as well use `` `$(<)' `` in place of `` `$<' ``.

How Patterns Match

A target pattern is composed of a `` `%' `` between a prefix and a suffix, either or both of which may be empty. The pattern matches a file name only if the file name starts with the prefix and ends with the suffix, without overlap. The text between the prefix and the suffix is called the **stem**. Thus, when the pattern `` `%.o' `` matches the file name `` `test.o' ``, the stem is `` `test' ``. The pattern rule prerequisites are turned into actual file names by substituting the stem for the character `` `%' ``. Thus, if in the same example one of the prerequisites is written as `` `%.c' ``, it expands to `` `test.c' ``.

When the target pattern does not contain a slash (and it usually does not), directory names in the file names are removed from the file name before it is compared with the target prefix and suffix. After the comparison of the file name to the target pattern, the directory names, along with the slash that ends them, are added on to the prerequisite file names generated from the pattern rule's prerequisite patterns and the file name. The directories are ignored only for the purpose of finding an implicit rule to use, not in the application of that rule. Thus, `` `e%t' `` matches the file name `` `src/eat' ``, with `` `src/a' `` as the stem. When prerequisites are turned into file names, the directories from the stem are added at the front, while the rest of the stem is substituted for the `` `%' ``. The stem `` `src/a' `` with a prerequisite pattern `` `c%r' `` gives the file name `` `src/car' ``.

Match-Anything Pattern Rules

When a pattern rule's target is just `` `%' ``, it matches any file name whatever. We call these rules **match-anything** rules. They are very useful, but it can take a lot of time for `make` to think about them, because it must consider every such rule for each file name listed either as a target or as a prerequisite.

Suppose the makefile mentions `` `foo.c' ``. For this target, `make` would have to consider making it by linking an object file `` `foo.c.o' ``, or by C compilation-and-linking in one step from `` `foo.c.c' ``, or by Pascal compilation-and-linking from `` `foo.c.p' ``, and many other possibilities.

We know these possibilities are ridiculous since `` `foo.c' `` is a C source file, not an executable. If `make` did consider these possibilities, it would ultimately reject them, because files such as `` `foo.c.o' `` and `` `foo.c.p' `` would not exist. But these possibilities are so numerous that `make` would run very slowly if it had to consider them.

To gain speed, we have put various constraints on the way `make` considers match-anything rules. There are two different constraints that can be applied, and each time you define a match-anything rule you must choose one or the other for that rule.

One choice is to mark the match-anything rule as **terminal** by defining it with a double colon. When a rule is terminal, it does not apply unless its prerequisites actually exist. Prerequisites that could be made with other implicit rules are not good enough. In other words, no further chaining is allowed beyond a terminal rule.

For example, the built-in implicit rules for extracting sources from RCS and SCCS files are terminal; as a result, if the file `` `foo.c,v' `` does not exist, `make` will not even consider trying to make it as an intermediate file from `` `foo.c,v.o' `` or from `` `RCS/SCCS/s.foo.c,v' ``. RCS and SCCS files are generally ultimate source files, which should not be remade from any other files; therefore, `make` can save time by not looking for ways to remake them.

If you do not mark the match-anything rule as terminal, then it is nonterminal. A nonterminal match-anything rule cannot apply to a file name that indicates a specific type of data. A file name indicates a specific type of data if some non-match-anything implicit rule target matches it.

For example, the file name `` `foo.c' `` matches the target for the pattern rule `` `%.c : %.y' `` (the rule to run Yacc). Regardless of whether this rule is actually applicable (which happens only if there is a file `` `foo.y' ``), the fact that its target matches is enough to prevent consideration of any nonterminal match-anything rules for the file `` `foo.c' ``. Thus, `make` will not even consider trying to make `` `foo.c' `` as an executable file from `` `foo.c.o' ``, `` `foo.c.c' ``, `` `foo.c.p' ``, etc.

The motivation for this constraint is that nonterminal match-anything rules are used for making files containing specific types of data (such as executable files) and a file name with a recognized suffix indicates some other specific type of data (such as a C source file).

Special built-in dummy pattern rules are provided solely to recognize certain file names so that nonterminal match-anything rules will not be considered. These dummy rules have no prerequisites and no commands, and they are ignored for all other purposes. For example, the built-in implicit rule

```
%.p :
```

exists to make sure that Pascal source files such as `` `foo.p' `` match a specific target pattern and thereby prevent time from being wasted looking for `` `foo.p.o' `` or `` `foo.p.c' ``.

Dummy pattern rules such as the one for `` `%.p' `` are made for every suffix listed as valid for use in suffix rules (see section Old-Fashioned Suffix Rules).

Canceling Implicit Rules

You can override a built-in implicit rule (or one you have defined yourself) by defining a new pattern rule with the same target and prerequisites, but different commands. When the new rule is defined, the built-in one is replaced. The new rule's position in the sequence of implicit rules is determined by where you write the new rule.

You can cancel a built-in implicit rule by defining a pattern rule with the same target and prerequisites, but no commands. For example, the following would cancel the rule that runs the assembler:

```
%.o : %.s
```

Defining Last-Resort Default Rules

You can define a last-resort implicit rule by writing a terminal match-anything pattern rule with no prerequisites (see section Match-Anything Pattern Rules). This is just like any other pattern rule; the only thing special about it is that it will match any target. So such a rule's commands are used for all targets and prerequisites that have no commands of their own and for which no other implicit rule applies.

For example, when testing a makefile, you might not care if the source files contain real data, only that they exist. Then you might do this:

```
%::
        touch $@
```

to cause all the source files needed (as prerequisites) to be created automatically.

You can instead define commands to be used for targets for which there are no rules at all, even ones which don't specify commands. You do this by writing a rule for the target `.DEFAULT`. Such a rule's commands are used for all prerequisites which do not appear as targets in any explicit rule, and for which no implicit rule applies. Naturally, there is no `.DEFAULT` rule unless you write one.

If you use `.DEFAULT` with no commands or prerequisites:

```
.DEFAULT:
```

the commands previously stored for `.DEFAULT` are cleared. Then `make` acts as if you had never defined `.DEFAULT` at all.

If you do not want a target to get the commands from a match-anything pattern rule or `.DEFAULT`, but you also do not want any commands to be run for the target, you can give it empty commands (see section Using Empty Commands).

You can use a last-resort rule to override part of another makefile. See section Overriding Part of Another Makefile.

Old-Fashioned Suffix Rules

Suffix rules are the old-fashioned way of defining implicit rules for `make`. Suffix rules are obsolete because pattern rules are more general and clearer. They are supported in GNU `make` for compatibility with old makefiles. They come in two kinds: **double-suffix** and **single-suffix**.

A double-suffix rule is defined by a pair of suffixes: the target suffix and the source suffix. It matches any file whose name ends with the target suffix. The corresponding implicit prerequisite is made by replacing the target suffix with the source suffix in the file name. A two-suffix rule whose target and source suffixes are `` `.o' `` and `` `.c' `` is equivalent to the pattern rule `` `%.o : %.c' ``.

A single-suffix rule is defined by a single suffix, which is the source suffix. It matches any file name, and the corresponding implicit prerequisite name is made by appending the source suffix. A single-suffix rule whose source suffix is `` `.c' `` is equivalent to the pattern rule `` `% : %.c' ``.

Suffix rule definitions are recognized by comparing each rule's target against a defined list of known suffixes. When `make` sees a rule whose target is a known suffix, this rule is considered a single-suffix rule. When `make` sees a rule whose target is two known suffixes concatenated, this rule is taken as a double-suffix rule.

For example, `` `.c' `` and `` `.o' `` are both on the default list of known suffixes. Therefore, if you define a rule whose target is `` `.c.o' ``, `make` takes it to be a double-suffix rule with source suffix `` `.c' `` and target suffix `` `.o' ``. Here is the old-fashioned way to define the rule for compiling a C source file:

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

Suffix rules cannot have any prerequisites of their own. If they have any, they are treated as normal files with funny names, not as suffix rules. Thus, the rule:

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

tells how to make the file `` `.c.o' `` from the prerequisite file `` `foo.h' ``, and is not at all like the pattern rule:

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

which tells how to make `` `.o' `` files from `` `.c' `` files, and makes all `` `.o' `` files using this pattern rule also depend on `` `foo.h' ``.

Suffix rules with no commands are also meaningless. They do not remove previous rules as do pattern rules with no commands (see section Canceling Implicit Rules). They simply enter the suffix or pair of suffixes concatenated as a target in the data base.

The known suffixes are simply the names of the prerequisites of the special target `.SUFFIXES`. You can add your own suffixes by writing a rule for `.SUFFIXES` that adds more prerequisites, as in:

```
.SUFFIXES: .hack .win
```

which adds `` `.hack' `` and `` `.win' `` to the end of the list of suffixes.

If you wish to eliminate the default known suffixes instead of just adding to them, write a rule for `.SUFFIXES` with no prerequisites. By special dispensation, this eliminates all existing prerequisites of `.SUFFIXES`. You can then write another rule to add the suffixes you want. For example,

```
.SUFFIXES:            # Delete the default suffixes
.SUFFIXES: .c .o .h   # Define our suffix list
```

The `` `-r' `` or `` `--no-builtin-rules' `` flag causes the default list of suffixes to be empty.

The variable `SUFFIXES` is defined to the default list of suffixes before `make` reads any makefiles. You can change the list of suffixes with a rule for the special target `.SUFFIXES`, but that does not alter this variable.

Implicit Rule Search Algorithm

Here is the procedure `make` uses for searching for an implicit rule for a target *t*. This procedure is followed for each double-colon rule with no commands, for each target of ordinary rules none of which have commands, and for each prerequisite that is not the target of any rule. It is also followed recursively for prerequisites that come from implicit rules, in the search for a chain of rules.

Suffix rules are not mentioned in this algorithm because suffix rules are converted to equivalent pattern rules once the makefiles have been read in.

For an archive member target of the form ``archive(member)'`, the following algorithm is run twice, first using the entire target name *t*, and second using ``(member)'` as the target *t* if the first run found no rule.

1. Split *t* into a directory part, called *d*, and the rest, called *n*. For example, if *t* is ``src/foo.o'`, then *d* is ``src/'` and *n* is ``foo.o'`.
2. Make a list of all the pattern rules one of whose targets matches *t* or *n*. If the target pattern contains a slash, it is matched against *t*; otherwise, against *n*.
3. If any rule in that list is *not* a match-anything rule, then remove all nonterminal match-anything rules from the list.
4. Remove from the list all rules with no commands.
5. For each pattern rule in the list:
    1. Find the stem *s*, which is the nonempty part of *t* or *n* matched by the ``%'` in the target pattern.
    2. Compute the prerequisite names by substituting *s* for ``%'`; if the target pattern does not contain a slash, append *d* to the front of each prerequisite name.
    3. Test whether all the prerequisites exist or ought to exist. (If a file name is mentioned in the makefile as a target or as an explicit prerequisite, then we say it ought to exist.) If all prerequisites exist or ought to exist, or there are no prerequisites, then this rule applies.
6. If no pattern rule has been found so far, try harder. For each pattern rule in the list:
    1. If the rule is terminal, ignore it and go on to the next rule.
    2. Compute the prerequisite names as before.
    3. Test whether all the prerequisites exist or ought to exist.
    4. For each prerequisite that does not exist, follow this algorithm recursively to see if the prerequisite can be made by an implicit rule.
    5. If all prerequisites exist, ought to exist, or can be made by implicit rules, then this rule applies.
7. If no implicit rule applies, the rule for `.DEFAULT`, if any, applies. In that case, give *t* the same commands that `.DEFAULT` has. Otherwise, there are no commands for *t*.

Once a rule that applies has been found, for each target pattern of the rule other than the one that matched *t* or *n*, the ``%'` in the pattern is replaced with *s* and the resultant file name is stored until the commands to remake the target file *t* are executed. After these commands are executed, each of these stored file names are entered into the data base and marked as having been updated and having the same update status as the file *t*.

When the commands of a pattern rule are executed for *t*, the automatic variables are set corresponding to the target and prerequisites. See section Automatic Variables.

Go to the first, previous, next, last section, table of contents.