GNU `make`

GNU `make`

This file documents the GNU `make` utility, which determines automatically which pieces of a large program need to be recompiled, and issues the commands to recompile them.

This is Edition 0.77, last updated 26 February 2023, of The GNU Make Manual, for GNU `make` version 4.4.1.

Table of Contents

Short Table of Contents

Next: An Introduction to Makefiles, Previous: GNU `make`, Up: GNU `make`   [Contents][Index]

1 Overview of `make`

The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. This manual describes GNU `make`, which was implemented by Richard Stallman and Roland McGrath. Development since Version 3.76 has been handled by Paul D. Smith.

GNU `make` conforms to section 6.2 of IEEE Standard 1003.2-1992 (POSIX.2).

Our examples show C programs, since they are most common, but you can use `make` with any programming language whose compiler can be run with a shell command. Indeed, `make` is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

- How to Read This Manual
- Problems and Bugs

Next: How to Read This Manual, Previous: Overview of `make`, Up: Overview of `make`   [Contents][Index]

Preparing

Preparing and Running Make

To prepare to use `make`, you must write a file called the *makefile* that describes the relationships among files in your program and provides commands for updating each file.

In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The `make` program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the recipes recorded in the data base.

You can provide command line arguments to `make` to control which files should be recompiled, or how. See How to Run `make`.

1.1 How to Read This Manual

If you are new to `make`, or are looking for a general introduction, read the first few sections of each chapter, skipping the later sections. In each chapter, the first few sections contain introductory or general information and the later sections contain specialized or technical information. The exception is the second chapter, An Introduction to Makefiles, all of which is introductory.

If you are familiar with other `make` programs, see Features of GNU `make`, which lists the enhancements GNU `make` has, and Incompatibilities and Missing Features, which explains the few things GNU `make` lacks that others have.

For a quick summary, see Summary of Options, Quick Reference, and Special Built-in Target Names.

1.2 Problems and Bugs

If you have problems with GNU `make` or think you've found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you've actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible makefile that reproduces the problem. Then send us the makefile and the exact results `make` gave you, including any error or warning messages. Please don't paraphrase these messages: it's best to cut and paste them into your report. When generating this small makefile, be sure to not use any non-free or unusual tools in your recipes: you can almost always emulate what such a tool would do with simple shell commands. Finally, be sure to explain what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you have a precise problem you can report it in one of two ways. Either send electronic mail to:

```
    bug-make@gnu.org
```

or use our Web-based project management tool, at:

```
    https://savannah.gnu.org/projects/make/
```

In addition to the information above, please be careful to include the version number of `make` you are using. You can get this information with the command '`make --version`'. Be sure also to include the type of machine and operating system you are using. One way to obtain this information is by looking at the final lines of output from the command '`make --help`'.

If you have a code change you'd like to submit, see the `README` file section "Submitting Patches" for information.

2 An Introduction to Makefiles

You need a file called a *makefile* to tell `make` what to do. Most often, the makefile tells `make` how to compile and link a program.

In this chapter, we will discuss a simple makefile that describes how to compile and link a text editor which consists of eight C source files and three header files. The makefile can also tell `make` how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation). To see a more complex example of a makefile, see Complex Makefile Example.

When `make` recompiles the editor, each changed C source file must be recompiled. If a header file has changed, each C source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

2.1 What a Rule Looks Like

A simple makefile consists of “rules” with the following shape:

```
target … : prerequisites …
        recipe
        …
        …
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘`clean`’ (see Phony Targets).

A *prerequisite* is a file that is used as input to create the target. A target often depends on several files.

A *recipe* is an action that `make` carries out. A recipe may have more than one command, either on the same line or each on its own line. **Please note:** you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary. If you prefer to prefix your recipes with a character other than tab, you can set the `.RECIPEPREFIX` variable to an alternate character (see Other Special Variables).

Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies a recipe for the target need not have prerequisites. For example, the rule containing the delete command associated with the target ‘`clean`’ does not have prerequisites.

A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the recipe on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action. See Writing Rules.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

2.2 A Simple Makefile

Here is a straightforward makefile that describes the way an executable file called `edit` depends on eight object files which, in turn, depend on eight C source and three header files.

In this example, all the C files include `defs.h`, but only those defining editing commands include `command.h`, and only low level files that change the editor buffer include `buffer.h`.

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

We split each long line into two lines using backslash/newline; this is like using one long line, but is easier to read. See Splitting Long Lines.

To use this makefile to create the executable file called `edit`, type:

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

In the example makefile, the targets include the executable file '`edit`', and the object files '`main.o`' and '`kbd.o`'. The prerequisites are files such as '`main.c`' and '`defs.h`'. In fact, each '`.o`' file is both a target and a prerequisite. Recipes include '`cc -c main.c`' and '`cc -c kbd.c`'.

When a target is a file, it needs to be recompiled or relinked if any of its prerequisites change. In addition, any prerequisites that are themselves automatically generated should be updated first. In this example, `edit` depends on each of the eight object files; the object file `main.o` depends on the source file `main.c` and on the header file `defs.h`.

A recipe may follow each line that contains a target and prerequisites. These recipes say how to update the target file. A tab character (or whatever character is specified by the `.RECIPEPREFIX` variable; see Other Special Variables) must come at the beginning of every line in the recipe to distinguish recipes from other lines in the makefile. (Bear in mind that `make` does not know anything about how the recipes work. It is up to you to supply recipes that will update the target file properly. All `make` does is execute the recipe you have specified when the target file needs to be updated.)

The target '`clean`' is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, '`clean`' is not a prerequisite of any other rule. Consequently, `make` never does anything with it unless you tell it specifically. Note that this rule not only is not a prerequisite, it also does not have any prerequisites, so the only purpose of the rule is to run the specified recipe. Targets that do not refer to files but are just actions are called *phony targets*. See Phony Targets, for information about this kind of target. See Errors in Recipes, to see how to cause `make` to ignore errors from `rm` or any other command.

Next: Variables Make Makefiles Simpler, Previous: A Simple Makefile, Up: An Introduction to Makefiles [Contents][Index]

2.3 How `make` Processes a Makefile

By default, `make` starts with the first target (not targets whose names start with '`.`' unless they also contain one or more '`/`'). This is called the *default goal*. (*Goals* are the targets that `make` strives ultimately to update. You can override this behavior using the command line (see Arguments to Specify the Goals) or with the `.DEFAULT_GOAL` special variable (see Other Special Variables).

In the simple example of the previous section, the default goal is to update the executable program `edit`; therefore, we put that rule first.

Thus, when you give the command:

```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit`; but before `make` can fully process this rule, it must process the rules for the files that `edit` depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each ‘`.o`’ file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as prerequisites of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not processed, unless you tell `make` to do so (with a command such as `make clean`).

Before recompiling an object file, `make` considers updating its prerequisites, the source file and header files. This makefile does not specify anything to be done for them—the ‘`.c`’ and ‘`.h`’ files are not the targets of any rules—so `make` does nothing for these files. But `make` would update automatically generated C programs, such as those made by Bison or Yacc, by their own rules at this time.

After recompiling whichever object files need it, `make` decides whether to relink `edit`. This must be done if the file `edit` does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than `edit`, so `edit` is relinked.

Thus, if we change the file `insert.c` and run `make`, `make` will compile that file to update `insert.o`, and then link `edit`. If we change the file `command.h` and run `make`, `make` will recompile the object files `kbd.o`, `command.o` and `files.o` and then link the file `edit`.

Next: Letting `make` Deduce the Recipes, Previous: How `make` Processes a Makefile, Up: An Introduction to Makefiles   [Contents][Index]

2.4 Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for `edit` (repeated here):

```
edit : main.o kbd.o command.o display.o \
              insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a variable. *Variables* allow a text string to be defined once and substituted in multiple places later (see How to Use Variables).

It is standard practice for every makefile to have a variable named `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, or `OBJ` which is a list of all object file names. We would define such a variable `objects` with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable’s value by writing ‘`$(objects)`’ (see How to Use Variables).

Here is how the complete simple makefile looks when you use a variable for the object files:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
```

```
clean :
        rm edit $(objects)
```

2.5 Letting `make` Deduce the Recipes

It is not necessary to spell out the recipes for compiling the individual C source files, because `make` can figure them out: it has an *implicit rule* for updating a '`.o`' file from a correspondingly named '`.c`' file using a '`cc -c`' command. For example, it will use the recipe '`cc -c main.c -o main.o`' to compile `main.c` into `main.o`. We can therefore omit the recipes from the rules for the object files. See Using Implicit Rules.

When a '`.c`' file is used automatically in this way, it is also automatically added to the list of prerequisites. We can therefore omit the '`.c`' files from the prerequisites, provided we omit the recipe.

Here is the entire example, with both of these changes, and a variable `objects` as suggested above:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        rm edit $(objects)
```

This is how we would write the makefile in actual practice. (The complications associated with '`clean`' are described elsewhere. See Phony Targets, and Errors in Recipes.)

Because implicit rules are so convenient, they are important. You will see them used frequently.

2.6 Another Style of Makefile

When the objects of a makefile are created only by implicit rules, an alternative style of makefile is possible. In this style of makefile, you group entries by their prerequisites instead of by their targets. Here is what one looks like:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here `defs.h` is given as a prerequisite of all the object files; `command.h` and `buffer.h` are prerequisites of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

2.7 Rules for Cleaning the Directory

Compiling a program is not the only thing you might want to write rules for. Makefiles commonly tell how to do a few other things besides compiling a program: for example, how to delete all the object files and executables so that the directory is '`clean`'.

Here is how we could write a `make` rule for cleaning our example editor:

```
clean:
        rm edit $(objects)
```

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. We would do this:

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

This prevents `make` from getting confused by an actual file called `clean` and causes it to continue in spite of errors from `rm`. (See Phony Targets, and Errors in Recipes.)

A rule such as this should not be placed at the beginning of the makefile, because we do not want it to run by default! Thus, in the example makefile, we want the rule for `edit`, which recompiles the editor, to remain the default goal.

Since `clean` is not a prerequisite of `edit`, this rule will not run at all if we give the command '`make`' with no arguments. In order to make the rule run, we have to type '`make clean`'. See How to Run `make`.

---

3 Writing Makefiles

The information that tells `make` how to recompile a system comes from reading a data base called the *makefile*.

3.1 What Makefiles Contain

Makefiles contain five kinds of things: *explicit rules*, *implicit rules*, *variable definitions*, *directives*, and *comments*. Rules, variables, and directives are described at length in later chapters.

- An *explicit rule* says when and how to remake one or more files, called the rule's *targets*. It lists the other files that the targets depend on, called the *prerequisites* of the target, and may also give a recipe to use to create or update the targets. See Writing Rules.
- An *implicit rule* says when and how to remake a class of files based on their names. It describes how a target may depend on a file with a name similar to the target and gives a recipe to create or update such a target. See Using Implicit Rules.
- A *variable definition* is a line that specifies a text string value for a variable that can be substituted into the text later. The simple makefile example shows a variable definition for `objects` as a list of all object files (see Variables Make Makefiles Simpler).
- A *directive* is an instruction for `make` to do something special while reading the makefile. These include:
  - Reading another makefile (see Including Other Makefiles).
  - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see Conditional Parts of Makefiles).
  - Defining a variable from a verbatim string containing multiple lines (see Defining Multi-Line Variables).
- '`#`' in a line of a makefile starts a *comment*. It and the rest of the line are ignored, except that a trailing backslash not escaped by another backslash will continue

the comment across multiple lines. A line containing just a comment (with perhaps spaces before it) is effectively blank, and is ignored. If you want a literal `#`, escape it with a backslash (e.g., `\#`). Comments may appear on any line in the makefile, although they are treated specially in certain situations.

You cannot use comments within variable references or function calls: any instance of `#` will be treated literally (rather than as the start of a comment) inside a variable reference or function call.

Comments within a recipe are passed to the shell, just as with any other recipe text. The shell decides how to interpret it: whether or not this is a comment is up to the shell.

Within a `define` directive, comments are not ignored during the definition of the variable, but rather kept intact in the value of the variable. When the variable is expanded they will either be treated as `make` comments or as recipe text, depending on the context in which the variable is evaluated.

- Splitting Long Lines

Previous: What Makefiles Contain, Up: What Makefiles Contain   [Contents][Index]

3.1.1 Splitting Long Lines

Makefiles use a “line-based” syntax in which the newline character is special and marks the end of a statement. GNU `make` has no limit on the length of a statement line, up to the amount of memory in your computer.

However, it is difficult to read lines which are too long to display without wrapping or scrolling. So, you can format your makefiles for readability by adding newlines into the middle of a statement: you do this by escaping the internal newlines with a backslash (`\`) character. Where we need to make a distinction we will refer to “physical lines” as a single line ending with a newline (regardless of whether it is escaped) and a “logical line” being a complete statement including all escaped newlines up to the first non-escaped newline.

The way in which backslash/newline combinations are handled depends on whether the statement is a recipe line or a non-recipe line. Handling of backslash/newline in a recipe line is discussed later (see Splitting Recipe Lines).

Outside of recipe lines, backslash/newlines are converted into a single space character. Once that is done, all whitespace around the backslash/newline is condensed into a single space: this includes all whitespace preceding the backslash, all whitespace at the beginning of the line after the backslash/newline, and any consecutive backslash/newline combinations.

If the `.POSIX` special target is defined then backslash/newline handling is modified slightly to conform to POSIX.2: first, whitespace preceding a backslash is not removed and second, consecutive backslash/newlines are not condensed.

Splitting Without Adding Whitespace

If you need to split a line but do *not* want any whitespace added, you can utilize a subtle trick: replace your backslash/newline pairs with the three characters dollar sign, backslash, and newline:

```
var := one$\
       word
```

After `make` removes the backslash/newline and condenses the following line into a single space, this is equivalent to:

```
var := one$ word
```

Then `make` will perform variable expansion. The variable reference ‘`$ `’ refers to a variable with the one-character name “ ” (space) which does not exist, and so expands to the empty string, giving a final assignment which is the equivalent of:

```
var := oneword
```

Next: Including Other Makefiles, Previous: What Makefiles Contain, Up: Writing Makefiles   [Contents][Index]

3.2 What Name to Give Your Makefile

By default, when `make` looks for the makefile, it tries the following names, in order: `GNUmakefile`, `makefile` and `Makefile`.

Normally you should call your makefile either `makefile` or `Makefile`. (We recommend `Makefile` because it appears prominently near the beginning of a directory listing, right near other important files such as `README`.) The first name checked, `GNUmakefile`, is not recommended for most makefiles. You should use this name if you have a makefile that is specific to GNU `make`, and will not be understood by other versions of `make`. Other `make` programs look for `makefile` and `Makefile`, but not `GNUmakefile`.

If `make` finds none of these names, it does not use any makefile. Then you must specify a goal with a command argument, and `make` will attempt to figure out how to remake it using only its built-in implicit rules. See Using Implicit Rules.

If you want to use a nonstandard name for your makefile, you can specify the makefile name with the '`-f`' or '`--file`' option. The arguments '`-f` *name*' or '`--file=`*name*' tell `make` to read the file *name* as the makefile. If you use more than one '`-f`' or '`--file`' option, you can specify several makefiles. All the makefiles are effectively concatenated in the order specified. The default makefile names `GNUmakefile`, `makefile` and `Makefile` are not checked automatically if you specify '`-f`' or '`--file`'.

Next: The Variable `MAKEFILES`, Previous: What Name to Give Your Makefile, Up: Writing Makefiles [Contents][Index]

3.3 Including Other Makefiles

The `include` directive tells `make` to suspend reading the current makefile and read one or more other makefiles before continuing. The directive is a line in the makefile that looks like this:

```
include filenames…
```

filenames can contain shell file name patterns. If *filenames* is empty, nothing is included and no error is printed.

Extra spaces are allowed and ignored at the beginning of the line, but the first character must not be a tab (or the value of `.RECIPEPREFIX`)—if the line begins with a tab, it will be considered a recipe line. Whitespace is required between `include` and the file names, and between file names; extra whitespace is ignored there and at the end of the directive. A comment starting with '`#`' is allowed at the end of the line. If the file names contain any variable or function references, they are expanded. See How to Use Variables.

For example, if you have three `.mk` files, `a.mk`, `b.mk`, and `c.mk`, and `$(bar)` expands to `bish bash`, then the following expression

```
include foo *.mk $(bar)
```

is equivalent to

```
include foo a.mk b.mk c.mk bish bash
```

When `make` processes an `include` directive, it suspends reading of the containing makefile and reads from each listed file in turn. When that is finished, `make` resumes reading the makefile in which the directive appears.

One occasion for using `include` directives is when several programs, handled by individual makefiles in various directories, need to use a common set of variable definitions (see Setting Variables) or pattern rules (see Defining and Redefining Pattern Rules).

Another such occasion is when you want to generate prerequisites from source files automatically; the prerequisites can be put in a file that is included by the main makefile. This practice is generally cleaner than that of somehow appending the prerequisites to the end of the main makefile as has been traditionally done with other versions of `make`. See Generating Prerequisites Automatically.

If the specified name does not start with a slash (or a drive letter and colon when GNU Make is compiled with MS-DOS / MS-Windows path support), and the file is not found in the current directory, several other directories are searched. First, any directories you have specified with the '`-I`' or '`--include-dir`' options are searched (see Summary of Options). Then the following directories (if they exist) are searched, in this order: *`prefix`*`/include` (normally `/usr/local/include` [1]) `/usr/gnu/include`, `/usr/local/include`, `/usr/include`.

The `.INCLUDE_DIRS` variable will contain the current list of directories that make will search for included files. See Other Special Variables.

You can avoid searching in these default directories by adding the command line option `-I` with the special value `-` (e.g., `-I-`) to the command line. This will cause `make` to forget any already-set include directories, including the default directories.

If an included makefile cannot be found in any of these directories it is not an immediately fatal error; processing of the makefile containing the `include` continues. Once it has finished reading makefiles, `make` will try to remake any that are out of date or don't exist. See How Makefiles Are Remade. Only after it has failed to find a rule to remake the makefile, or it found a rule but the recipe failed, will `make` diagnose the missing makefile as a fatal error.

If you want `make` to simply ignore a makefile which does not exist or cannot be remade, with no error message, use the `-include` directive instead of `include`, like this:

```
-include filenames…
```

This acts like `include` in every way except that there is no error (not even a warning) if any of the *filenames* (or any prerequisites of any of the *filenames*) do not exist or cannot be remade.

For compatibility with some other `make` implementations, `sinclude` is another name for `-include`.

3.4 The Variable `MAKEFILES`

If the environment variable `MAKEFILES` is defined, `make` considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive: various directories are searched for those files (see Including Other Makefiles). In addition, the default goal is never taken from one of these makefiles (or any makefile included by them) and it is not an error if the files listed in `MAKEFILES` are not found.

The main use of `MAKEFILES` is in communication between recursive invocations of `make` (see Recursive Use of `make`). It usually is not desirable to set the environment variable before a top-level invocation of `make`, because it is usually better not to mess with a makefile from outside. However, if you are running `make` without a specific makefile, a makefile in `MAKEFILES` can do useful things to help the built-in implicit rules work better, such as defining search paths (see Searching Directories for Prerequisites).

Some users are tempted to set `MAKEFILES` in the environment automatically on login, and program makefiles to expect this to be done. This is a very bad idea, because such makefiles will fail to work if run by anyone else. It is much better to write explicit `include` directives in the makefiles. See Including Other Makefiles.

3.5 How Makefiles Are Remade

Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want `make` to get an up-to-date version of the makefile to read in.

To this end, after reading in all makefiles `make` will consider each as a goal target, in the order in which they were processed, and attempt to update it. If parallel builds (see Parallel Execution) are enabled then makefiles will be rebuilt in parallel as well.

If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see Using Implicit Rules), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, `make` starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.) Each restart will cause the special variable `MAKE_RESTARTS` to be updated (see Other Special Variables).

If you know that one or more of your makefiles cannot be remade and you want to keep `make` from performing an implicit rule search on them, perhaps for efficiency reasons, you can use any normal method of preventing implicit rule look-up to do so. For example, you can write an explicit rule with the makefile as the target, and an empty recipe (see Using Empty Recipes).

If the makefiles specify a double-colon rule to remake a file with a recipe but no prerequisites, that file will always be remade (see Double-Colon Rules). In the case of makefiles, a makefile that has a double-colon rule with a recipe but no prerequisites will be remade every time `make` is run, and then again after `make` starts over and reads the makefiles in again. This would cause an infinite loop: `make` would constantly remake the makefile and restart, and never do anything else. So, to avoid this, `make` will **not** attempt to remake makefiles which are specified as targets of a double-colon rule with a recipe but no prerequisites.

Phony targets (see Phony Targets) have the same effect: they are never considered up-to-date and so an included file marked as phony would cause `make` to restart continuously. To avoid this `make` will not attempt to remake makefiles which are marked phony.

You can take advantage of this to optimize startup time: if you know you don't need your `Makefile` to be remade you can prevent make from trying to remake it by adding either:

```
.PHONY: Makefile
```

or:

```
Makefile:: ;
```

If you do not specify any makefiles to be read with '`-f`' or '`--file`' options, `make` will try the default makefile names; see What Name to Give Your Makefile. Unlike makefiles explicitly requested with '`-f`' or '`--file`' options, `make` is not certain that these makefiles should exist. However, if a default makefile does not exist but can be created by running `make` rules, you probably want the rules to be run so that the makefile can be used.

Therefore, if none of the default makefiles exists, `make` will try to make each of them until it succeeds in making one, or it runs out of names to try. Note that it is not an error if `make` cannot find or make any makefile; a makefile is not always necessary.

When you use the '`-t`' or '`--touch`' option (see Instead of Executing Recipes), you would not want to use an out-of-date makefile to decide which targets to touch. So the '`-t`' option has no effect on updating makefiles; they are really updated even if '`-t`' is specified. Likewise, '`-q`' (or '`--question`') and '`-n`' (or '`--just-print`') do not prevent updating of makefiles, because an out-of-date makefile would result in the wrong output for other targets. Thus, '`make -f mfile -n foo`' will update `mfile`, read it in, and then print the recipe to update `foo` and its prerequisites without running it. The recipe printed for `foo` will be the one specified in the updated contents of `mfile`.

However, on occasion you might actually wish to prevent updating of even the makefiles. You can do this by specifying the makefiles as goals in the command line as well as specifying them as makefiles. When the makefile name is specified explicitly as a goal, the options '`-t`' and so on do apply to them.

Thus, '`make -f mfile -n mfile foo`' would read the makefile `mfile`, print the recipe needed to update it without actually running it, and then print the recipe needed to update `foo` without running that. The recipe for `foo` will be the one specified by the existing contents of `mfile`.

Next: How `make` Reads a Makefile, Previous: How Makefiles Are Remade, Up: Writing Makefiles [Contents][Index]

3.6 Overriding Part of Another Makefile

Sometimes it is useful to have a makefile that is mostly just like another makefile. You can often use the '`include`' directive to include one in the other, and add more targets or variable definitions. However, it is invalid for two makefiles to give different recipes for the same target. But there is another way.

In the containing makefile (the one that wants to include the other), you can use a match-anything pattern rule to say that to remake any target that cannot be made from the information in the containing makefile, `make` should look in another makefile. See Defining and Redefining Pattern Rules, for more information on pattern rules.

For example, if you have a makefile called `Makefile` that says how to make the target '`foo`' (and other targets), you can write a makefile called `GNUmakefile` that contains:

```
foo:
        frobnicate > foo

%: force
        @$(MAKE) -f Makefile $@
force: ;
```

If you say '`make foo`', `make` will find `GNUmakefile`, read it, and see that to make `foo`, it needs to run the recipe '`frobnicate > foo`'. If you say '`make bar`', `make` will find no way to make `bar` in `GNUmakefile`, so it will use the recipe from the pattern rule: '`make -f Makefile bar`'. If `Makefile` provides a rule for updating `bar`, `make` will apply the rule. And likewise for any other target that `GNUmakefile` does not say how to make.

The way this works is that the pattern rule has a pattern of just '`%`', so it matches any target whatever. The rule specifies a prerequisite `force`, to guarantee that the recipe will be run even if the target file already exists. We give the `force` target an empty recipe to prevent `make` from searching for an implicit rule to build it—otherwise it would apply the same match-anything rule to `force` itself and create a prerequisite loop!

Next: How Makefiles Are Parsed, Previous: Overriding Part of Another Makefile, Up: Writing Makefiles [Contents][Index]

3.7 How `make` Reads a Makefile

GNU `make` does its work in two distinct phases. During the first phase it reads all the makefiles, included makefiles, etc. and internalizes all the variables and their values and implicit and explicit rules, and builds a dependency graph of all the targets and their prerequisites. During the second phase, `make` uses this internalized data to determine which targets need to be updated and run the recipes necessary to update them.

It's important to understand this two-phase approach because it has a direct impact on how variable and function expansion happens; this is often a source of some confusion when writing makefiles. Below is a summary of the different constructs that can be found in a makefile, and the phase in which expansion happens for each part of the construct.

We say that expansion is *immediate* if it happens during the first phase: `make` will expand that part of the construct as the makefile is parsed. We say that expansion is *deferred* if it is not immediate. Expansion of a deferred construct part is delayed until the expansion is used: either when it is referenced in an immediate context, or when it is needed during the second phase.

You may not be familiar with some of these constructs yet. You can reference this section as you become familiar with them, in later chapters.

Variable Assignment

Variable definitions are parsed as follows:

```
immediate = deferred
immediate ?= deferred
immediate := immediate
immediate ::= immediate
immediate :::= immediate-with-escape
immediate += deferred or immediate
immediate != immediate

define immediate
  deferred
endef

define immediate =
  deferred
endef

define immediate ?=
  deferred
endef

define immediate :=
  immediate
endef

define immediate ::=
  immediate
endef

define immediate :::=
  immediate-with-escape
endef

define immediate +=
  deferred or immediate
endef

define immediate !=
  immediate
endef
```

For the append operator '`+=`', the right-hand side is considered immediate if the variable was previously set as a simple variable ('`:=`' or '`::=`'), and deferred otherwise.

For the *immediate-with-escape* operator '`:::=`', the value on the right-hand side is immediately expanded but then escaped (that is, all instances of `$` in the result of the expansion are replaced with `$$`).

For the shell assignment operator '`!=`', the right-hand side is evaluated immediately and handed to the shell. The result is stored in the variable named on the left, and that variable is considered a recursively expanded variable (and will thus be re-evaluated on each reference).

Conditional Directives

Conditional directives are parsed immediately. This means, for example, that automatic variables cannot be used in conditional directives, as automatic variables are not set until the recipe for that rule is invoked. If you need to use automatic variables in a conditional directive you *must* move the condition into the recipe and use shell conditional syntax instead.

Rule Definition

A rule is always expanded the same way, regardless of the form:

```
immediate : immediate ; deferred
        deferred
```

That is, the target and prerequisite sections are expanded immediately, and the recipe used to build the target is always deferred. This is true for explicit rules, pattern rules, suffix rules, static pattern rules, and simple prerequisite definitions.

3.8 How Makefiles Are Parsed

GNU `make` parses makefiles line-by-line. Parsing proceeds using the following steps:

1. Read in a full logical line, including backslash-escaped lines (see Splitting Long Lines).
2. Remove comments (see What Makefiles Contain).
3. If the line begins with the recipe prefix character and we are in a rule context, add the line to the current recipe and read the next line (see Recipe Syntax).
4. Expand elements of the line which appear in an *immediate* expansion context (see How `make` Reads a Makefile).
5. Scan the line for a separator character, such as '`:`' or '`=`', to determine whether the line is a macro assignment or a rule (see Recipe Syntax).
6. Internalize the resulting operation and read the next line.

An important consequence of this is that a macro can expand to an entire rule, *if it is one line long*. This will work:

```
myrule = target : ; echo built

$(myrule)
```

However, this will not work because `make` does not re-split lines after it has expanded them:

```
define myrule
target:
        echo built
endef

$(myrule)
```

The above makefile results in the definition of a target '`target`' with prerequisites '`echo`' and '`built`', as if the makefile contained `target: echo built`, rather than a rule with a recipe. Newlines still present in a line after expansion is complete are ignored as normal whitespace.

In order to properly expand a multi-line macro you must use the `eval` function: this causes the `make` parser to be run on the results of the expanded macro (see The `eval` Function).

3.9 Secondary Expansion

Previously we learned that GNU `make` works in two distinct phases: a read-in phase and a target-update phase (see How `make` Reads a Makefile). GNU Make also has the ability to enable a *second expansion* of the prerequisites (only) for some or all targets defined in the makefile. In order for this second expansion to occur, the special target `.SECONDEXPANSION` must be defined before the first prerequisite list that makes use of this feature.

If `.SECONDEXPANSION` is defined then when GNU `make` needs to check the prerequisites of a target, the prerequisites are expanded a *second time*. In most circumstances this secondary expansion will have no effect, since all variable and function references will have been expanded during the initial parsing of the makefiles. In order to take advantage of the secondary expansion phase of the parser, then, it's necessary to *escape* the variable or function reference in the makefile. In this case the first expansion merely un-escapes the reference but doesn't expand it, and expansion is left to the secondary expansion phase. For example, consider this makefile:

```
.SECONDEXPANSION:
ONEVAR = onefile
TWOVAR = twofile
myfile: $(ONEVAR) $$(TWOVAR)
```

After the first expansion phase the prerequisites list of the `myfile` target will be `onefile` and `$(TWOVAR)`; the first (unescaped) variable reference to *ONEVAR* is expanded, while the second (escaped) variable reference is simply unescaped, without being recognized as a variable reference. Now during the secondary expansion the first word is expanded again but since it contains no variable or function references it remains the value `onefile`, while the second word is now a normal reference to the variable *TWOVAR*, which is expanded to the value `twofile`. The final result is that there are two prerequisites, `onefile` and `twofile`.

Obviously, this is not a very interesting case since the same result could more easily have been achieved simply by having both variables appear, unescaped, in the prerequisites list. One difference becomes apparent if the variables are reset; consider this example:

```
.SECONDEXPANSION:
AVAR = top
onefile: $(AVAR)
twofile: $$(AVAR)
AVAR = bottom
```

Here the prerequisite of `onefile` will be expanded immediately, and resolve to the value `top`, while the prerequisite of `twofile` will not be full expanded until the secondary expansion and yield a value of `bottom`.

This is marginally more exciting, but the true power of this feature only becomes apparent when you discover that secondary expansions always take place within the scope of the automatic variables for that target. This means that you can use variables such as `$@`, `$*`, etc. during the second expansion and they will have their expected values, just as in the recipe. All you have to do is defer the expansion by escaping the `$`. Also, secondary expansion occurs for both explicit and implicit (pattern) rules. Knowing this, the possible uses for this feature increase dramatically. For example:

```
.SECONDEXPANSION:
main_OBJS := main.o try.o test.o
lib_OBJS := lib.o api.o

main lib: $$($$@_OBJS)
```

Here, after the initial expansion the prerequisites of both the `main` and `lib` targets will be `$($@_OBJS)`. During the secondary expansion, the `$@` variable is set to the name of the target and so the expansion for the `main` target will yield `$(main_OBJS)`, or `main.o try.o test.o`, while the secondary expansion for the `lib` target will yield `$(lib_OBJS)`, or `lib.o api.o`.

You can also mix in functions here, as long as they are properly escaped:

```
main_SRCS := main.c try.c test.c
lib_SRCS := lib.c api.c

.SECONDEXPANSION:
main lib: $$(patsubst %.c,%.o,$$($$@_SRCS))
```

This version allows users to specify source files rather than object files, but gives the same resulting prerequisites list as the previous example.

Evaluation of automatic variables during the secondary expansion phase, especially of the target name variable `$$@`, behaves similarly to evaluation within recipes. However, there are some subtle differences and "corner cases" which come into play for the different types of rule definitions that `make` understands. The subtleties of using the different automatic variables are described below.

Secondary Expansion of Explicit Rules

During the secondary expansion of explicit rules, `$$@` and `$$%` evaluate, respectively, to the file name of the target and, when the target is an archive member, the target member name. The `$$<` variable evaluates to the first prerequisite in the first rule for this target. `$$^` and `$$+` evaluate to the list of all prerequisites of rules *that have already appeared* for the same target (`$$+` with repetitions and `$$^` without). The following example will help illustrate these behaviors:

```
.SECONDEXPANSION:

foo: foo.1 bar.1 $$< $$^ $$+    # line #1

foo: foo.2 bar.2 $$< $$^ $$+    # line #2

foo: foo.3 bar.3 $$< $$^ $$+    # line #3
```

In the first prerequisite list, all three variables (`$$<`, `$$^`, and `$$+`) expand to the empty string. In the second, they will have values `foo.1`, `foo.1 bar.1`, and `foo.1 bar.1` respectively. In the third they will have values `foo.1`, `foo.1 bar.1 foo.2 bar.2`, and `foo.1 bar.1 foo.2 bar.2 foo.1 foo.1 bar.1 foo.1 bar.1` respectively.

Rules undergo secondary expansion in makefile order, except that the rule with the recipe is always evaluated last.

The variables `$$?` and `$$*` are not available and expand to the empty string.

Secondary Expansion of Static Pattern Rules

Rules for secondary expansion of static pattern rules are identical to those for explicit rules, above, with one exception: for static pattern rules the `$$*` variable is set to the pattern stem. As with explicit rules, `$$?` is not available and expands to the empty string.

Secondary Expansion of Implicit Rules

As `make` searches for an implicit rule, it substitutes the stem and then performs secondary expansion for every rule with a matching target pattern. The value of the automatic variables is derived in the same fashion as for static pattern rules. As an example:

```
.SECONDEXPANSION:

foo: bar

foo foz: fo%: bo%

%oo: $$< $$^ $$+ $$*
```

When the implicit rule is tried for target `foo`, `$$<` expands to `bar`, `$$^` expands to `bar boo`, `$$+` also expands to `bar boo`, and `$$*` expands to `f`.

Note that the directory prefix (D), as described in Implicit Rule Search Algorithm, is appended (after expansion) to all the patterns in the prerequisites list. As an example:

```
.SECONDEXPANSION:

/tmp/foo.o:

%.o: $$(addsuffix /%.c,foo bar) foo.h
        @echo $^
```

The prerequisite list printed, after the secondary expansion and directory prefix reconstruction, will be `/tmp/foo/foo.c /tmp/bar/foo.c foo.h`. If you are not interested in this reconstruction, you can use `$$*` instead of `%` in the prerequisites list.

Next: Writing Recipes in Rules, Previous: Writing Makefiles, Up: GNU `make`   [Contents][Index]

4 Writing Rules

A *rule* appears in the makefile and says when and how to remake certain files, called the rule’s *targets* (most often only one per rule). It lists the other files that are the *prerequisites* of the target, and the *recipe* to use to create or update the target.

The order of rules is not significant, except for determining the *default goal*: the target for `make` to consider, if you do not otherwise specify one. The default goal is the first target of the first rule in the first makefile. There are two exceptions: a target starting with a period is not a default unless it also contains one or more slashes, ‘`/`’; and, a target that defines a pattern rule has no effect on the default goal. (See Defining and Redefining Pattern Rules.)

Therefore, we usually write the makefile so that the first rule is the one for compiling the entire program or all the programs described by the makefile (often with a target called ‘`all`’). See Arguments to Specify the Goals.

4.1 Rule Example

Here is an example of a rule:

```
foo.o : foo.c defs.h       # module for twiddling the frobs
        cc -c -g foo.c
```

Its target is `foo.o` and its prerequisites are `foo.c` and `defs.h`. It has one command in the recipe: ‘`cc -c -g foo.c`’. The recipe starts with a tab to identify it as a recipe.

This rule says two things:

- How to decide whether `foo.o` is out of date: it is out of date if it does not exist, or if either `foo.c` or `defs.h` is more recent than it.
- How to update the file `foo.o`: by running `cc` as stated. The recipe does not explicitly mention `defs.h`, but we presume that `foo.c` includes it, and that is why `defs.h` was added to the prerequisites.

4.2 Rule Syntax

In general, a rule looks like this:

```
targets : prerequisites
        recipe
        …
```

or like this:

```
targets : prerequisites ; recipe
        recipe
        …
```

The *targets* are file names, separated by spaces. Wildcard characters may be used (see Using Wildcard Characters in File Names) and a name of the form `a(m)` represents member *m* in archive file *a* (see Archive Members as Targets). Usually there is only one target per rule, but occasionally there is a reason to have more (see Multiple Targets in a Rule).

The *recipe* lines start with a tab character (or the first character in the value of the `.RECIPEPREFIX` variable; see Other Special Variables). The first recipe line may appear on the line after the prerequisites, with a tab character, or may appear on the same line, with a semicolon. Either way, the effect is the same. There are other differences in the syntax of recipes. See Writing Recipes in Rules.

Because dollar signs are used to start `make` variable references, if you really want a dollar sign in a target or prerequisite you must write two of them, '`$$`' (see How to Use Variables). If you have enabled secondary expansion (see Secondary Expansion) and you want a literal dollar sign in the prerequisites list, you must actually write *four* dollar signs ('`$$$$`').

You may split a long line by inserting a backslash followed by a newline, but this is not required, as `make` places no limit on the length of a line in a makefile.

A rule tells `make` two things: when the targets are out of date, and how to update them when necessary.

The criterion for being out of date is specified in terms of the *prerequisites*, which consist of file names separated by spaces. (Wildcards and archive members (see Using `make` to Update Archive Files) are allowed here too.) A target is out of date if it does not exist or if it is older than any of the prerequisites (by comparison of last-modification times). The idea is that the contents of the target file are computed based on information in the prerequisites, so if any of the prerequisites changes, the contents of the existing target file are no longer necessarily valid.

How to update is specified by a *recipe*. This is one or more lines to be executed by the shell (normally '`sh`'), but with some extra features (see Writing Recipes in Rules).

Next: Using Wildcard Characters in File Names, Previous: Rule Syntax, Up: Writing Rules [Contents] [Index]

4.3 Types of Prerequisites

There are two different types of prerequisites understood by GNU `make`: normal prerequisites, described in the previous section, and *order-only* prerequisites. A normal prerequisite makes two statements: first, it imposes an order in which recipes will be invoked: the recipes for all prerequisites of a target will be completed before the recipe for the target is started. Second, it imposes a dependency relationship: if any prerequisite is newer than the target, then the target is considered out-of-date and must be rebuilt.

Normally, this is exactly what you want: if a target's prerequisite is updated, then the target should also be updated.

Occasionally you may want to ensure that a prerequisite is built before a target, but *without* forcing the target to be updated if the prerequisite is updated. *Order-only* prerequisites are used to create this type of relationship. Order-only prerequisites can be specified by placing a pipe symbol (`|`) in the prerequisites list: any prerequisites to the left of the pipe symbol are normal; any prerequisites to the right are order-only:

```
targets : normal-prerequisites | order-only-prerequisites
```

The normal prerequisites section may of course be empty. Also, you may still declare multiple lines of prerequisites for the same target: they are appended appropriately (normal prerequisites are appended to the list of normal prerequisites; order-only prerequisites are appended to the list of order-only prerequisites). Note that if you declare the same file to be both a normal and an order-only prerequisite, the normal prerequisite takes precedence (since they have a strict superset of the behavior of an order-only prerequisite).

Order-only prerequisites are never checked when determining if the target is out of date; even order-only prerequisites marked as phony (see Phony Targets) will not cause the target to be rebuilt.

Consider an example where your targets are to be placed in a separate directory, and that directory might not exist before `make` is run. In this situation, you want the directory to be created before any targets are placed into it but, because the timestamps on directories change whenever a file is added, removed, or renamed, we certainly don't want to rebuild all the targets whenever the directory's timestamp changes. One way to manage this is with order-only prerequisites: make the directory an order-only prerequisite on all the targets:

```
OBJDIR := objdir
OBJS := $(addprefix $(OBJDIR)/,foo.o bar.o baz.o)

$(OBJDIR)/%.o : %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<

all: $(OBJS)

$(OBJS): | $(OBJDIR)
```