GNU Make

A Program for Directing Recompilation

GNU `make` Version 3.79

April 2000

Richard M. Stallman and Roland McGrath

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.

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 commands recorded in the data base.

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

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 section 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.

What a Rule Looks Like

A simple makefile consists of "rules" with the following shape:

```
target ... : prerequisites ...
        command
        ...
        ...
```

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 section Phony Targets).

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

A **command** is an action that `make` carries out. A rule may have more than one command, each on its own line. **Please note:** you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies commands 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 commands on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action. See section 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.

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.

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. Commands 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 shell command follows each line that contains a target and prerequisites. These shell commands say how to update the target file. A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that make does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All make does is execute the commands in the rule 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 commands. Targets that do not refer to files but are just actions are called **phony targets**. See section [Phony Targets](), for information about this kind of target. See section [Errors in Commands](), to see how to cause make to ignore errors from rm or any other command.

How make Processes a Makefile

By default, make starts with the first target (not targets whose names start with `.'). This is called the **default goal**. (**Goals** are the targets that make strives ultimately to update. See section [Arguments to Specify the Goals]().)

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' ``.

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 section [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 section [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)
```

Letting `make` Deduce the Commands

It is not necessary to spell out the commands 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 command `cc -c main.c -o main.o` to compile `main.c` into `main.o`. We can therefore omit the commands from the rules for the object files. See section 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 commands.

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 section [Phony Targets](#), and section [Errors in Commands](#).)

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

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.

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 section [Phony Targets](#), and section [Errors in](#)

.)

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 section How to Run `make`.

Writing Makefiles

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

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, call the **prerequisites** of the target, and may also give commands to use to create or update the targets. See section 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 commands to create or update such a target. See section 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 section Variables Make Makefiles Simpler).

?   A **directive** is a command for `make` to do something special while reading the makefile. These include:

?   Reading another makefile (see section Including Other Makefiles).

?   Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see section Conditional Parts of Makefiles).

?   Defining a variable from a verbatim string containing multiple lines (see section Defining Variables Verbatim).

?   `` `#' `` 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. Comments may appear on any of the lines in the makefile, except within a `define` directive, and perhaps within commands (where the shell decides what is a comment). A line containing just a comment (with perhaps spaces before it) is effectively blank, and is ignored.

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 section 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' ``.