Copyright (C) 2025 Ignat Grishechko. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ``GNU Free Documentation License''.

Content

[1. Terms, definitions, and symbols. 5](#_Toc189069984)

[2. Quick introduction. 6](#_Toc189069985)

[2.1. What a Rule Looks Like. 6](#_Toc189069986)

[2.2. A Simple Makefile. 8](#_Toc189069987)

[2.3. Phony Targets. 9](#_Toc189069988)

[3. Using variables. 10](#_Toc189069989)

[3.1. Introduction. 10](#_Toc189069990)

[3.2. Automatic variables. 10](#_Toc189069991)

[4. Using builtin functions. 12](#_Toc189069992)

[4.1. Function Call Syntax. 12](#_Toc189069993)

[4.2. Simpler functions. 12](#_Toc189069994)

[4.3. Complex functions. 17](#_Toc189069995)

[5. Writing rules. 18](#_Toc189069996)

[5.1. Target. 18](#_Toc189069997)

[5.2. Prerequisites. 22](#_Toc189069998)

[5.3. Recipe. 23](#_Toc189069999)

[5.4. Elementary example. 23](#_Toc189070000)

[5.5. Elementary example. 24](#_Toc189070001)

[5.6. Elementary example. 24](#_Toc189070002)

[5.7. Elementary example. 24](#_Toc189070003)

[6. 25](#_Toc189070004)

[7. Thoughts and recommendations. 26](#_Toc189070005)

[7.1. “Must have” 26](#_Toc189070006)

[7.2. Choosing shell. 27](#_Toc189070007)

[7.3. Used shell commands. 27](#_Toc189070008)

[8. Examples. 27](#_Toc189070009)

[9. Experiments. 27](#_Toc189070010)

[9.1. Separated compilation. 27](#_Toc189070011)

1. **Terms, definitions, and symbols**.

*Program code* – the sequence of instructions, that that computers can follow to perform tasks.

*Makefile* – the main file of the gnu make build system. Contain the *program code*, that build system will run.

1. Quick introduction.
   1. What a Rule Looks Like.

Code listing 1

* target … : prerequisites …
* recipe…
  + 1. Makefile.

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

A simple *makefile* consists of “*rules*” with the following shape (see *Code listing 1*)

* + 1. Target.
       1. As file name.

A *target* is usually the name of a file that is generated by a program.

* + - 1. As “phony” name.

A *target* can also be the name of an action to carry out, such as ‘clean’.

* + 1. Prerequisite.

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

* + 1. Recipe.

A recipe is in a rule with or without prerequisites and with serves to create a target file if any of the prerequisites change.

Please note: you need to put a tab character or another character which was specified in .RECIPEPREFIX variable at the beginning of every recipe line.

* 1. A Simple Makefile*.*
* edit : main.o files.o
* main.o : main.c
* cc -c main.c
* files.o : files.c
* cc -c files.c
* clean :
* rm edit main.o files.o utils.o

Code listing 2

* + 1. Auto building of the target.

Pay attention – the rule named edit (see Code listing 2) probably automatically build the target named edit from the source objects files noted in prerequisites.

* 1. Phony Targets.
* .PHONY: clean
* clean:
* rm \*.o temp

Code listing 3

* + 1. What is it?

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.

If you write a rule whose recipe will not create the target file, the recipe will be executed every time the target comes up for remaking.

A target will not work properly if a file with the same name will be created in the same directory as makefile if you will not mark such target as .PHONY (see Code listing 3).

1. Using variables.
   1. Introduction.

Variables and functions in all parts of a makefile are expanded when read, except for in recipes, the right-hand sides of variable definitions using ‘=’, and the bodies of variable definitions using the define directive. The value a variable expands to is that of its most recent definition at the time of expansion.

* 1. Automatic variables.
     1. $@

The file name of the target of the rule. If the target is an archive member, then ‘$@’ is the name of the archive file. In a pattern rule that has multiple targets ‘$@’ is the name of whichever target caused the rule’s recipe to be run.

* + 1. $%

The target member name, when the target is an archive member.

* + 1. $<

The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule.

* + 1. $?

The names of all the prerequisites that are newer than the target, with spaces between them. If the target does not exist, all prerequisites will be included. For prerequisites which are archive members, only the named member is used.

* + 1. $^

The names of all the prerequisites, with spaces between them. For prerequisites which are archive members, only the named member is used. This list does **not** contain any of the order-only prerequisites.

* + 1. $+

This is like ‘$^’, but prerequisites listed more than once are duplicated in the order they were listed in the makefile.

* + 1. $|

The names of all the order-only prerequisites, with spaces between them.

* + 1. $\*

The stem with which an implicit rule matches (see 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. You should generally avoid using ‘$\*’ except in implicit rules or static pattern rules.

1. Using builtin functions.
   1. Function Call Syntax.
   2. Simpler functions.
      1. Text functions.
         1. $(subst from,to,text)

Performs a textual replacement on the text text: each occurrence of from is replaced by to.

* + - 1. $(patsubst pattern,replacement,text)

Finds whitespace-separated words in text that match pattern and replaces them with replacement. Here pattern may contain a ‘%’ which acts as a wildcard, matching any number of any characters within a word. If replacement also contains a ‘%’, the ‘%’ is replaced by the text that matched the ‘%’ in pattern. Words that do not match the pattern are kept without change in the output. Only the first ‘%’ in the pattern and replacement is treated this way; any subsequent ‘%’ is unchanged.

* + - 1. $(strip string)

Removes leading and trailing whitespace from string and replaces each internal sequence of one or more whitespace characters with a single space.

* + - 1. $(findstring find,in)

Searches in for an occurrence of find. If it occurs, the value is find; otherwise, the value is empty.

* + - 1. $(filter pattern…,text)

Returns all whitespace-separated words in text that do match any of the pattern words, removing any words that do not match. The patterns are written using ‘%’, just like the patterns used in the patsubst function.

* + - 1. $(filter-out pattern…,text)

Returns all whitespace-separated words in text that do not match any of the pattern words, removing the words that do match one or more. This is the exact opposite of the filter function.

* + - 1. $(sort list)

Sorts the words of list in lexical order, removing duplicate words. The output is a list of words separated by single spaces.

* + - 1. $(word n,text)

Returns the nth word of text. The legitimate values of n start from 1. If n is bigger than the number of words in text, the value is empty.

* + - 1. $(wordlist s,e,text)

Performs a textual replacement on the text text: each occurrence of from is replaced by to.

* + - 1. $(words text)

Returns the number of words in text.

* + - 1. $(firstword names…)

The argument names is regarded as a series of names, separated by whitespace.

* + - 1. $(lastword names…)

The argument names is regarded as a series of names, separated by whitespace.

* + 1. File functions.
       1. $(dir names…).

Extracts the directory-part of each file name in names.

* + - 1. $(notdir names…)

Extracts all but the directory-part of each file name in names.

* + - 1. $(suffix names…)

Extracts the suffix of each file name in names.

* + - 1. $(basename names…)

Extracts all but the suffix of each file name in names.

* + - 1. $(addsuffix suffix,names…)

The argument names is regarded as a series of names, separated by whitespace; suffix is used as a unit. The value of suffix is appended to the end of each individual name.

* + - 1. $(addprefix prefix,names…)

The argument names is regarded as a series of names, separated by whitespace; prefix is used as a unit. The value of prefix is prepended to the front of each individual name.

* + - 1. $(join list1,list2)

Concatenates the two arguments word by word: the two first words (one from each argument) concatenated form the first word of the result, the two second words form the second word of the result, and so on.

* + - 1. $(wildcard pattern)

The argument pattern is a file name pattern, typically containing wildcard characters (as in shell file name patterns).

* + - 1. $(realpath names…)

For each file name in names return the canonical absolute name. A canonical name does not contain any . or .. components, nor any repeated path separators (/) or symlinks.

* + - 1. $(abspath names…)

For each file name in names return an absolute name that does not contain any . or .. components, nor any repeated path separators (/). Note that, in contrast to realpath function, abspath does not resolve symlinks and does not require the file names to refer to an existing file or directory.

* + 1. Conditional functions.
       1. Syntax of Conditionals.

Code listing 4

* conditional-directive-one
* text-if-one-is-true
* else conditional-directive-two
* text-if-two-is-true
* else
* text-if-one-and-two-are-false
* endif
  + - 1. $(if condition,then-part[,else-part])

Performs a textual replacement on the text text: each occurrence of from is replaced by to.

* 1. Complex functions.

1. Writing rules.
   1. Target.
      1. Special Built-in Target Names.
         1. .PHONY

The prerequisites of the special target .PHONY are considered to be phony targets. When it is time to consider such a target, make will run its recipe unconditionally, regardless of whether a file with that name exists or what its last-modification time is.

* + - 1. .SUFFIXES

The prerequisites of the special target .SUFFIXES are the list of suffixes to be used in checking for 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.

* + - 1. .DEFAULT

The recipe specified for .DEFAULT is used for any target for which no rules are found (either explicit rules or implicit rules). If a .DEFAULT recipe is specified, every file mentioned as a prerequisite, but not as a target (because the .DEFAULT is a target itself) in a rule, will have that recipe executed on its behalf. See Implicit Rule Search Algorithm.

Note: MAKE AN EXAMPLE

* + - 1. .PRECIOUS

The targets which .PRECIOUS depends on are given the following special treatment: if make is killed or interrupted during the execution of their recipes, the target file is not deleted. Also, if the target is an intermediate file, it will not be deleted after it is no longer needed, as is normally done. It overlaps with the .SECONDARY special target.

* + - 1. .INTERMEDIATE

The prerequisites what are also targets which .INTERMEDIATE depends on are treated as intermediate files. .INTERMEDIATE with no prerequisites has no effect.

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 creates b and then updates the target from b. But if b is an intermediate file, then make won’t create b unless one of its prerequisites is out of date. This means the target depending on b won’t be rebuilt either, unless 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’ command showing which file it is deleting.

* + - 1. .NOTINTERMEDIATE

Prerequisites of the special target .NOTINTERMEDIATE are never considered intermediate files. .NOTINTERMEDIATE with no prerequisites causes all targets to be treated as not intermediate. If the prerequisite is a target pattern then targets that are built using that pattern rule are not considered intermediate.

* + - 1. .SECONDARY

The targets which .SECONDARY depends on are treated as intermediate files, except that they are never automatically deleted. See Chains of Implicit Rules.

* + - 1. .SECONDEXPANSION

If .SECONDEXPANSION is mentioned as a target anywhere in the *makefile*, then all prerequisite lists defined after it appears will be expanded a second time after all *makefiles* have been read in. See Secondary Expansion.

* + - 1. .DELETE\_ON\_ERROR

If .DELETE\_ON\_ERROR is mentioned as a target anywhere in the makefile, then make will delete the target of a rule if it has changed and its recipe exits with a nonzero exit status, just as it does when it receives a signal. See Errors in Recipes.

* + - 1. .IGNORE

If you specify prerequisites for .IGNORE, then make will ignore errors in execution of the recipe for those particular files. The recipe for .IGNORE (if any) is ignored.

If mentioned as a target with no prerequisites, .IGNORE says to ignore errors in execution of recipes for all files. This usage of ‘.IGNORE’ is supported only for historical compatibility. Since this affects every recipe in the makefile, it is not very useful; we recommend you use the more selective ways to ignore errors in specific recipes. See Errors in Recipes.

* + - 1. .LOW\_RESOLUTION\_TIME

If you specify prerequisites for .LOW\_RESOLUTION\_TIME, make assumes that these files are created by commands that generate low resolution time stamps. The recipe for the .LOW\_RESOLUTION\_TIME target are ignored.

The high resolution file time stamps of many modern file systems lessen the chance of make incorrectly concluding that a file is up to date. Unfortunately, some hosts do not provide a way to set a high resolution file time stamp, so commands like ‘cp -p’ that explicitly set a file’s time stamp must discard its sub-second part. If a file is created by such a command, you should list it as a prerequisite of .LOW\_RESOLUTION\_TIME so that make does not mistakenly conclude that the file is out of date. For example:

* + - 1. .LOW\_RESOLUTION\_TIME

Since ‘cp -p’ discards the sub-second part of src’s time stamp, dst is typically slightly older than src even when it is up to date. The .LOW\_RESOLUTION\_TIME line causes make to consider dst to be up to date if its time stamp is at the start of the same second that src’s time stamp is in.

Due to a limitation of the archive format, archive member time stamps are always low resolution. You need not list archive members as prerequisites of .LOW\_RESOLUTION\_TIME, as make does this automatically.

* + - 1. .SILENT

If you specify prerequisites for .SILENT, then make will not print the recipe used to remake those particular files before executing them. The recipe for .SILENT is ignored.

If mentioned as a target with no prerequisites, .SILENT says not to print any recipes before executing them. You may also use more selective ways to silence specific recipe command lines. See Recipe Echoing. If you want to silence all recipes for a particular run of make, use the ‘-s’ or ‘--silent’ option (see Summary of Options).

* + - 1. .EXPORT\_ALL\_VARIABLES

Simply by being mentioned as a target, this tells make to export all variables to child processes by default. This is an alternative to using export with no arguments. See Communicating Variables to a Sub-make.

* + - 1. .NOTPARALLEL

If .NOTPARALLEL is mentioned as a target with no prerequisites, all targets in this invocation of make will be run serially, even if the ‘-j’ option is given. Any recursively invoked make command will still run recipes in parallel (unless its makefile also contains this target).

If .NOTPARALLEL has targets as prerequisites, then all the prerequisites of those targets will be run serially. This implicitly adds a .WAIT between each prerequisite of the listed targets. See Disabling Parallel Execution.

* + - 1. .ONESHELL

If .ONESHELL is mentioned as a target, then when a target is built all lines of the recipe will be given to a single invocation of the shell rather than each line being invoked separately. See Recipe Execution.

* + - 1. .POSIX

If .POSIX is mentioned as a target, then the makefile will be parsed and run in POSIX-conforming mode. This does not mean that only POSIX-conforming makefiles will be accepted: all advanced GNU make features are still available. Rather, this target causes make to behave as required by POSIX in those areas where make’s default behavior differs.

In particular, if this target is mentioned then recipes will be invoked as if the shell had been passed the -e flag: the first failing command in a recipe will cause the recipe to fail immediately.

* 1. Prerequisites.
  2. Recipe.
  3. Elementary example.

Code listing 4

* foo.o : foo.c defs.h ; # module for twiddling the frobs
* cc -c -g foo.c
  + 1. Meaning.

This rule says two things:

* How to decide whether foo.o is out of date.
* 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.
  + 1. Composition.
       1. Target.

The targets are file names, separated by spaces.

A single file name can specify many files using wildcard characters. The wildcard characters in make are ‘\*’, ‘?’ and ‘[…]’, the same as in the Bourne shell. For example, \*.c specifies a list of all the files (in the working directory) whose names end in ‘.c’.

The character ‘~’ at the beginning of a file name also has special significance. If alone, or followed by a slash, it represents your home directory. For example ~/bin expands to /home/you/bin. If the ‘~’ is followed by a word, the string represents the home directory of the user named by that word.

Wildcard expansion happens automatically in rules. But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function. If you want to do wildcard expansion in such places, you need to use the wildcard function, like this:

* $(wildcard pattern…)
  1. Elementary example.
  2. Elementary example.
  3. Elementary example.

2. Thoughts and recommendations.
   1. “Must have”
      1. .PRESIOUS.

You can also list the target pattern of an implicit rule (such as ‘%.o’) as a prerequisite file of the special target .PRECIOUS to preserve intermediate files created by rules whose target patterns match that file’s name

NEED AN EXPERIMENT

* + 1. Intermediate files.

Why should you have intermediate files? Because if the *make* process was killed in the process of building then files will be built as corrupted. And the one corrupted file will create an another corrupted file with a very high probability during the next build attempt. Maybe you can use them for the debug process?

* + 1. Targets as prerequisites.

You CAN use targets as prerequisites for another rules because any TARGET is a FILE in the end.

* + 1. Shell variable problems.

Every Makefile should contain this line:

* SHELL = /bin/sh

This is made to avoid trouble on systems where the *SHELL* variable might be inherited from the environment.

* + 1. Suffixes.

Different *make* programs have incompatible suffix lists and implicit rules, and this sometimes creates confusion or misbehavior. So it is a good idea to set the suffix list explicitly using only the suffixes you need in the particular Makefile, like this:

* .SUFFIXES:
* .SUFFIXES: .c .o

The first line clears out the suffix list, the second introduces all suffixes which may be subject to implicit rules in this Makefile.

* + 1. Suffixes.
  1. Choosing shell.

More common shells are more preferable. In over words – write for sh but not for csh or for some other shells.

* 1. Used shell commands.

All used shell commands are recommended to be wrapped into variables, listed in the top part of the makefile. This is will make the porting process much more easy.

1. Examples.
2. Experiments.
   1. Separated compilation.
      1. Description.

Sometimes we do not want to use a fully constructed compiler and use preprocessor separately from other phases of compiler.

* + 1. Experiment process.

HERE WILL BE VERY SCARY EXPERIMENT

Bibliography.

**Текущий документ не содержит источников.**