PEL Topic Index

GNU Make

See also: Pl - Make	GNU Make tools:	GNU Autotools @ Wikipedia, GNU Coding Standard, section 7, Filesystem Hierarchy Standard (FHS 3.0)
	GNU Make Manuals :	• GNU Make Top page • How to run make • GNU Make - Appendix A - Quick Reference • Makefile Conventions • Autoconf Portable Make Programming

GNU Make Rules

Including Other Makefiles

Include makefiles	`include filenames…`	`-include filenames…`	Use the -include so that make ignores a makefile which does not exist or cannot be remade, with no error message.
GNU Make Escaping	`dollar := $$  pound := \#`		

GNU Make Rules

(See section on **implicit rules** below)

Topic	Rule syntax format		Description	
Rule Syntax	targets : prerequisites recipe …		• Multiple line recipe, the on mostly used. • The recipe lines must start with a **TAB** character (or the string identified by the .RECIPEPREFIX pseudo-variable.	
	targets : prerequisites ; recipe recipe …		• It is also possible to to identify a recipe on the same line as the prerequisites, separated from them by a semicolon. • This allow writing a single-line rule.	

Wildcards	Wildcards can be used in targets and prerequisites.
	• They are expanded in target and prerequisites
	• They are **not** expanded in variable definitions:
	  • See **wildcard examples**
	  • But **wildcard functions** can be use to expand in variable definition as in: `objects := $(wildcard *.o)`
			*****	All files, like '*.c'

			?	Expand to characters
			[…]	
			~	At beginning of path name, like ~/bin expands to your home bin directory
			~user	Expands the the home directory of specific user
Searching directories The Basics: VPATH and vpath	**VPATH**	The value of the VPATH make variable specifies a list of directories that make should search. • Each directory in the list can be separated by space or : • On MS-DOS, Windows: space or ;	Example: `VPATH = src:../headers`	
Selective search Use vpath to find sources, not targets.	**vpath** directive	Same as VPATH but more selective: only applies to a particular class of file names. The path statement format is one of the 3 forms. The last 2 clear search path for the specified scope (file pattern or all): • **vpath *pattern directories*** *set search of* pattern to directories • **vpath *pattern*** *clear search path for specified pattern* • **vpath** *clear search path for all scopes*	The first form sets the directory search for a specified file name pattern, like the following: `vpath %.h ../headers`	

Directory search for Link Libraries	Note: that make treats prerequisites of the form **-l*name*** as library names. The -lname is expanded to the full path of the library name with starts with the 'lib' prefix. For example:
	`foo : foo.c -lcurses`
	`  cc $^ -o $@`
	
	will cause the following command to be executed if needed:
	`cc foo.c /usr/lib/libcurses.a -o foo`
	
	This behaviour is customizable by the **.LIBPATTERNS** special variable.			

Phony Targets
See also:
• Rules without Recipes or Prerequisites
• Empty target files to record events
	• A phony target is a target that is not really the name of a file, it's just a name for a recipe to be executed when you make an explicit request.
	• Use it to avoid a conflict with the name of a file, and to improve performance: implicit rule search is skipped for .PHONY targets.
	  • Example:
	`.PHONY: clean`
	`clean:`
	`  rm *.o temp`
	• Some older make versions did not support .PHONY , so a FORCE target without receipt or prerequisite was used:
	`FORCE:`
	
	• Also useful for recursive makes processing multiple directories with loops, and other case. See the GNU manual			

Special Built-in Targets	These include: **.PHONY .SUFFIXES .DEFAULT .PRECIOUS .INTERMEDIATE .SECONDARY .SECONDEXPANSION .DELETE_ON_ERROR .IGNORE .LOW_RESOLUTION_TIME .SILENT .EXPORT_ALL_VARIABLES .NOTPARALLEL .ONESHELL .POSIX .FEATURES**			
Other Special Variables	**MAKEFILE_LIST .DEFAULT_GOAL MAKE_RESTART MAKE_TERMOUT MAKE_TERMERR .RECIPEPREFIX .VARIABLES .FEATURES .INCLUDE_DIRS .EXTRA_PREREQ**			

GNU Make Recipes

Recipe line 1st char	suppress echoing with: @	Ignore recipe line error with: -	Prevent "instead of execution", marks the line as "recursive" ensure the line is executed even when make is invoked with the -n -t or -q command line option, with: +		
Recipe execution	By default: each recipe line is executed in a new sub-shell	Use one shell for all lines with: .ONESHELL:	• Select a shell with: SHELL • Shell arguments with: .SHELLFLAGS		
Recursive make • export and unexport directives.	Variable CURDIR : pathname of current directory	• Use variable MAKE to recurse make. • Variable MAKEFLAGS pass make flags to the sub-make.	• Variable MAKEFILES is exported if set to anything: set to space-separated names of make files. • It's also possible to export or un-export a specific variable with the export and unexport directives.		
Communicating options to sub-make	This section describe the use of the following variables: MAKEFLAGS, MAKEOVERRIDES, MFLAGS and GNUMAKEFLAGS,				

Canned Recipes	Define *"canned"* recipe with the **define** statement:	`define run-yacc =`
		`yacc $(firstword $^)`
		`mv y.tab.c $@`
		`endef`
			It can then be used later as in:	`foo.c : foo.y`
				`  $(run-yacc)`	

Empty Recipes	A recipe that does nothing. For example:	`target: ;`	Used to:	• Prevent a target from getting implicit recipes • Avoid errors for targets that will be created as side-effect of another recipe	

GNU Make Conditionals

Conditional syntax
See also:
conditional example
	`ifeq (arg1, arg2)`
	`ifeq 'arg1' 'arg2'`
	`ifeq "arg1" "arg2"`
	`ifeq "arg1" 'arg2'`
	`ifeq 'arg1' "arg2"`
		`ifneq (arg1, arg2)`
		`ifneq 'arg1' 'arg2'`
		`ifneq "arg1" "arg2"`
		`ifneq "arg1" 'arg2'`
		`ifneq 'arg1' "arg2"`
			`ifdef variable-name`	`ifndef variable-name`	`else`
					`else` conditional
					`endif`

GNU Make Text Transforming Functions

Function Call Syntax	Format	Arguments	Style
	• `$(function arguments)` • `${function arguments}`	• separated from the function name by 1 or more spaces or tabs • arguments are separated by commas	Use the same style of delimited () or {} inside the entire expression.

Text Functions	`$(subst from,to,text)`
	`$(patsubst pattern,replacement,text)`
	Alternative to `patsubst` is **Substitution References** of the form:
	• `$(var:a=b)`
	• `${var:a=b}`
		`$(strip string)`
		`$(findstring find,in)`
		`$(filter pattern…,text)`
		`$(filter-out pattern…,text)`
		`$(sort list)`
			`$(word n,text)`
			`$(wordlist s,e,text)`
			`$(words text)`
			`$(firstword names…)`
			`$(lastword names…)`

File Name Functions	For each of these functions the argument is regarded as a series of file names, separated by whitespace. Each file name in the series is transformed the same way and the results are concatenated with single spaces between them.		

	`$(dir names…)`
	`$(notdir names…)`
	`$(suffix names…)`
		`$(basename names…)`
		`$(addsuffix suffix,names…)`
		`$(addprefix prefix,names…)`
			`$(join list1,list2)`
			`$(wildcard pattern)`
			`$(realpath names…)`
			`$(abspath names…)`

Conditional Functions	`$(if condition,then-part[,else-part])`	`$(or condition1[,condition2[,condition3…]])`	`$(and condition1[,condition2[,condition3…]])`

The foreach Function	`$(foreach var,list,text)`	An example of this is show next:	`dirs := a b c d` `files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))`

The file Function	`$(file op filename[,text])`	Used to read or write from a file. For example, the following write commands to execute in a temporary command file that it executes then deletes:	`program: $(OBJECTS)`
			`$(file >$@.in,$^)`
			`$(CMD) $(CMDFLAGS) @$@.in`
			`@rm $@.in`

The call Function	`$(call variable,param,param,…)`	The following example reverses the arguments:	`reverse = $(2) $(1)`
			
			`foo = $(call reverse,a,b)`

		This sets variable LS to the path of the path of the ls program, something like /bin/ls	`pathsearch = $(firstword $(wildcard $(addsuffix /$(1),$ (subst :, ,$(PATH)))))`
			
			`LS := $(call pathsearch,ls)`

The value Function	`$(value variable)`	Provides a way to use the value of a variable without having it expanded.		
The eval Function	`$(eval expression)`			
The origin Function	`$(origin variable)`	Returns how the variable was defined. It can return one of the following: undefined, default, environment, environment override, file, command line, override, automatic.		
The flavour Function	`$(flavor variable)`	Returns the flavour of the variable. It can be one of the following: undefined, recursive, simple.		
Functions that control Make	These functions control the way Make runs and are used to provide information to the user.	`$(error text…)`	`$(warning text…)`	`$(info text…)`
The shell Function	The shell function performs command expansion similar to what backquote does in the shell. • After the `$(shell …)` execution, the exit status is placed inside the .SHELLSTATUS variable. • See the following examples:		To set the contents variable wit h a space separating each line: `contents := $(shell cat foo)`	Set files to a space separated list of C file names: `files := $(shell echo *.c)`
The guile Function	If GNU Make is built with Guile support the .FEATURES variable includes the word *guile*. The guile function is then available. Make expands its argument then it is passed to Guile for evaluation. See **GNU Guile Integration**.			

GNU Make Implicit Rules

Implicit Rule Topic	Description

Using Implicit Rules	• To use them refrain from writing the recipe for a kind of target.
	• Each implicit rule has a target and prerequisite patterns.
	• Write a rule to identify extra prerequisites like header files prerequisites to an object file.
	• There may be several implicit rules for the same target (for example a rule to generate object file from C files, another rule to generate object file from C++ files).
	• See the **catalogue of built-in-rules**. It is possible to **cancel an implicit rule**.
	• Make searches for implicit rules for:
	  • each target that has no recipe,
	  • each double-colon rule that has no recipe,
	  • a file that is only mentioned as a prerequisite.
	  • The **Implicit Rule Search Algorithm** describes how the search for an implicit rule is done.
	• A **chain of implicit rules** can be used to make the target from a prerequisite. But only one instance of an implicit rule can only be used in the chain.
	• It's possible to define **last-resort default rules** to **override part of another makefile**.
	• To prevent an implicit rule to apply to a specific target create an **empty recipe** for that target.

• **Pattern Rules**	Example:
	`%o : %c`
	`recipe`
		The example pattern rule says how to make *stem*.o from another file *stem*.c
		• Expansions using '%' in pattern occurs after any variable and function expansion.
		• More than one pattern rule may match a target: make will choose the "best fit" rule. See **How Pattern Match**.

Special GNU Make Variables

Make Goals	**MAKECMDGOALS**	This variable is set to the list of targets (goals) specified in the command line. If there were none, the variable is empty.

Variables used in Implicit Rules

Variable Name	Description	Default value	Flag Variable	Description and default value (if any)
AR	Archive-maintaining program	ar	**ARFLAGS**	Flags to give the archive-maintaining program; default **'rv'**
AS	Program for compiling assembly files	as	**ASFLAGS**	Extra flags to give to the assembler (when explicitly invoked on a '.s' or '.S' file)
CC	Program for compiling C files	cc	**CFLAGS**	Extra flags to give to the C compiler.
CXX	Program for compiling C++ files	g++	**CXXFLAGS**	Extra flags to give to the C++ compiler.
CPP	Program for running the C preprocessor, with results to standard output	$(CC) -E	**CPPFLAGS**	Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers).
FC	Program for compiling or preprocessing Fortran and Ratfor files	f77	**FFLAGS**	Extra flags to give to the Fortran compiler.
			RFLAGS	Extra flags to give to the Fortran compiler for Ratfor files.
M2C	Program to compile Modula-2 files	m2c		
PC	Program to compile Pascal files	pc	**PFLAGS**	Extra flags to give to the Pascal compiler.
CO	Program for extracting a file from RCS	co	**COFLAGS**	Extra flags to give to the RCS co program.
GET	Program for extracting a file from SCCS	get	**GFLAGS**	Extra flags to give to the SCCS get program.
LEX	Program to use to turn Lex grammars into source code	lex	**LFLAGS**	Extra flags to give to Lex.
YACC	Program to use to turn Yacc grammars into source code	yacc	**YFLAGS**	Extra flags to give to Yacc.
LINT	Program to use to run lint on source code	lint	**LINTFLAGS**	Extra flags to give to lint.
MAKEINFO	Program to convert a Texinfo source file into an Info file	makeinfo		
TEX	Program to make TeX DVI files from TeX source	tex		
TEXI2DVI	Program to make TeX DVI files from Texinfo source	texi2dvi		
WEAVE	Program to translate Web into TeX	weave		

CWEAVE	Program to translate C Web into TeX	weave		
TANGLE	Program to translate Web into Pascal	tangle		
CTANGLE	Program to translate C Web into C	tangle		
RM	Command to remove a file	rm -f		
			LDFLAGS	Extra flags to give to compilers when they are supposed to invoke the linker, 'ld', such as -L. Libraries (-lfoo) should be added to the LDLIBS variable instead.
			LDLIBS	Library flags or names given to compilers when they are supposed to invoke the linker, 'ld'. Non-library linker flags, such as -L, should go in the LDFLAGS variable.
			LOADLIBES	Deprecated (but still supported) alternative to LDLIBS.

Automatic Variable	**Expands to**	**Notes and examples**
$@	File name of the **target**. For archive(member): name or **archive**.	
$(@D)	The **directory** part of the target	If the target is just a file name, then the value of $(@D) is **.**
$(@F)	The **file name** (with extension) of the target	
$%	File name of target archive **member**	
$(%D)	The **directory** part of the target archive member	
$(%F)	The **file name** (with extension) of the target archive member	
$<	Name of the first **prerequisite**	
$(<D)	The **directory** part of the prerequisite	
$(<F)	The **file name** (with extension) of the prerequisite	
$?	Names of **all prerequisites newer than target** with spaces between them. • For archive(member), only contain the member.	Also useful in explicit rules when the receipt must operate on only the prerequisites that have changed.
$(?D)	List of the **directory** part of all prerequisites newer than target	
$(?F)	List of the **file name** (with extension) of all prerequisites newer than target	
$^	The names **of all prerequisites** with spaces between them. • For archive(member), only contain the member. • No duplicates in the list	Does not contain order-only prerequisites.
$(^D)	List of the **directory** part of all prerequisites (no duplicates)	
$(^F)	Lis of the **file name** (with extension) of all prerequisites (no duplicates)	
$+	The names **of all prerequisites** with spaces between them. • For archive(member), only contain the member. • **Duplicates are allowed** in the list in the same order as received	Useful when linking where it might be required to repeat the name of a library
$(+D)	List of the **directory** part of all prerequisites (with duplicates)	
$(+F)	List of the **file name** (with extension) of all prerequisites (with duplicates)	
$|	The names of **all order-only prerequisites** with spaces between them.	
$*	• For implicit rule: the **stem** which an implicit rule matches. • For explicit rule, there is no *stem* : expands to the target name minus the suffix.	• Implicit rule: if target is *dir/a.foo.b* and the target pattern is *a.%.b* then the stem is *dir/foo* • Explicit rule: If target is *foo.c*, then $* expands to *foo*.
$(*D)	The **directory** part of the stem	
$(*F)	The **file name** (with extension) of the stem	

Suffix Rules - Obsolete Old-fashioned Suffix Rules

Kinds of old-fashioned suffix rule	**Example of suffix rule**	**Corresponding pattern rule**	**Description**
double-suffix	.c.o	**%.o : %.c**	Matches any file whose name ends with the target suffix.
single-suffix	.c	**% : %.c**	Matches any file name, and the corresponding implicit prerequisite name is made by appending the source suffix
	The old-fashioned suffix rules are obsolete because the pattern rules are more general and clearer. • Suffix rules cannot have any prerequisites of their own. • Suffix sure without recipe are meaningless.		

Assignment operators

OP	**Description**	**Example**
	Rules	
:		non-terminal
::	Makes the rule terminal: it's prerequisite may not be an intermediate file.	
	Using Variables	

=	Non-terminal recursively expanded variable assignment.
	See:
	• **The two-flavours of Variables**
	• **Setting Variables**
		The following will echo Huh?:
		`foo = $(bar)`
		`bar = $(ugh)`
		`ugh = Huh?`
		
		`all:;echo $(foo)`

:=	Simply expanded variables
	See:
	• **The two-flavours of Variables**
		The following:
		`x := foo`
		`y := $(x) bar`
		`x := later`
		
		is equivalent to:
		`y := foo bar`
		`x := later`

::=	Simply expanded variables - 2012 POSIX standard compliant.
	See:
	• **The two-flavours of Variables**
		The following:
		`x ::= foo`
		`y ::= $(x) bar`
		`x ::= later`
		
		is equivalent to:
		`y ::= foo bar`
		`x ::= later`

OP	Description	Example

?=	Set variable if it is not already set.
	See:
	• **Setting Variables**
		The following:
		`FOO ?= bar`
		
		is equivalent to:
		`ifeq ($(origin FOO), undefined)`
		`FOO = bar`
		`endif`

!=	Shell assignment operator: used to execute a shell script and set a variable to its output.
	See:
	• **Setting Variables**
	
	Note that after the != execution, the exit status is placed inside the .SHELLSTATUS variable.
		For example, if you don't expect a $ character to be part of the output string:
		`hash != printf '\043'`
		`file_list != find . -name '*.c'`
		
		If you expect $ character(s) to be part of the output, then it's better to use another form:
		`hash := $(shell printf '\043')`
		`var := $(shell find . -name "*.c")`

+=	**Append text to a variable**
	The text append operation is affected by the flavour of the original variable assignment (by = or := operators.)
		The following:
		`objects = main.o foo.o bar.o utils.o`
		`objects += another.o`
		is equivalent to:
		`objects = main.o foo.o bar.o utils.o`
		`objects := $(objects) another.o`

	The **Override Directive** : how to set a variable in the make file even if the user has set it with a command argument.
	
	Appending More Text To Variables
	
	Defining Multi-Line Variables
		To override a variable that might have been set in the command line:
		`override variable = value`
		or
		`override variable := value`
		
		To append more text to a variable defined on the command line:
		`override variable += more text`
		
		It's also possible to override directives with define directive:
		`override define foo =`
		`bar`
		`endef`