↖ Topic Index

Emacs support for Make Files

Description	Keystroke	Function	Note
Make support	• Emacs natively supports several Make dialect modes as listed below. • PEL adds several commands and user-options that add control to the editing behaviour. See: • **pel-modes-activating-superword-mode** : PEL automatically activates super-word-mode for make files. Use **<f11> t <f2>** to access the customization group.		
Open this PDF file. See also: ∑ **Help/Info**	**<f11> SPC M <f1>** **<f12> <f1>**	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **𝔓 - Make** local PDF. If the prefix argument (like **C-u** or **M--**) is used, then it opens the remote GitHub hosted raw PDF instead. If the **pel-flip-help-pdf-arg** user-option is set it's the other way around.
∑ **Customize** PEL make support	**<f11> SPC M <f2>** **<f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL make support: **pel-use-makefile** • **pel-make-mode-alist** to identify more file regexp and a make file major mode that must be used for those files. • **pel-makefile-activates-minor-modes** lists minor modes to automatically activate in makefile major modes. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
𝔓 - Make	**<f11> SPC M <f3>** **<f12> <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs makefile support: makefile. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Select Make dialect mode See also: • ∑ **Customize** • ∑ **File/Directory Variables**	Emacs supports several dialects of **make**. It automatically selects the dialect when a file is visited using the mode and file specification association identified in the **auto-mode-alist** variable. The support associates the name and extensions of most make files with the corresponding dialect mode. The following make file dialect modes are supported: • makefile-mode (the based mode upon which all following modes are derived): • makefile-automake-mode : .am • makefile-bsdmake-mode : [Mm]akefile, .mk, .make • makefile-gmake-mode : GNUmakefile • makefile-imake-mode : Imakefile • makefile-makepp-mode : .makepp • makefile-nmake-mode : .mak PEL implements the makefile-nmake-mode to support Microsoft NMAKE syntax. • Some projects use the .mak extension for their makefile (the **dmd project** for example). • With PEL, set up the association using the **pel-auto-mode-alist** user-option. • You can access the relevant customization buffer for this user-option by using PEL **<f11> <f2> p** key sequence. See ∑ **Customize** • Its also possible to use file variables to explicitly identify the make dialect mode: write something like this on the first line: `-*- mode: makefile-gmake; -*-` • You can also use the following commands to manually activate one of these modes when on of them is already active.		
Activate automake mode	• **C-c RET C-a** • **C-c C-m C-a**	(**makefile-automake-mode**)	Activates the **automake** mode • The mode-line lighter is : Makefile.am
Activate BSD make mode	• **C-c RET C-b** • **C-c C-m C-b**	(**makefile-bsdmake-mode**)	Activates the **BSD make** mode. • BSD Make is the default make on macOS and BSD OS systems. • The mode-line lighter is : BSDmakefile
Activate GNU make mode	• **C-c RET C-g** • **C-c C-m C-g**	(**makefile-gmake-mode**)	Activates the **GNU make** mode. • The mode-line lighter is : GNUmakefile ⚠ Because this key sequence ends with **C-g**, type the **Esc** key 3 times to escape from the C-c C-m prefix. You can also use a key not in the list.
Activate imake mode	• **C-c RET <tab>** • **C-c C-m C-i**	(**makefile-imake-mode**)	Activate the **imake** mode • The mode-line lighter is : Imakefile
Activate standard make mode	• **C-c RET RET** • **C-c C-m C-m**	(**makefile-mode**)	Activates the major mode for editing standard Makefiles. • The mode-line lighter is : Makefile
Activate makepp mode	• **C-c RET C-p** • **C-c C-m C-p**	(**makefile-makepp-mode**)	Activates the **makepp** mode. Also called **make++** • makepp is written in Perl. It is mostly useful for writing C++ specific make files, as it expands GNU Make and removes the requirement of using recursive make. • The mode-line lighter is : Makeppfile
Activate NMAKE mode	• **C-c RET C-n** • **C-c C-m C-n**	(**makefile-nmake-mode**)	Activates the nmake mode, supporting Microsoft's NMAKE makefile syntax. • The mode-line lighter is: Nmake
Navigate	The standard Emacs make-mode.el package provides the 2 commands to navigate across make target/dependency statements. PEL complements this with commands to navigate across the macro definition statements.		
beginning of next token See also: ∑ **Navigation**	**C-<right>**	(**pel-forward-token-start** &optional N)	Move to the beginning of next word/symbol. • It handles characters that may be part of symbol in the current major mode, and jumps over them but stops at whitespace and operators. • Supports numerical argument for repetition. • Negative argument reverses the movement direction. ☛Shift marking works with this command. ☝Useful when the superword-mode is not activated: allows jumping to next symbol while the word commands stop at each word separator character.
beginning of previous token See also: ∑ **Navigation**	**C-<left>**	(**pel-backward-token-start** &optional N)	Move to the beginning of previous word/symbol. • It handles characters that may be part of symbol in the current major mode (like '_' in C), and jumps over them but stops at whitespace and operators. • Supports numerical argument for repetition. • Negative argument reverses the movement direction. ☛Shift marking works with this command. ☝Useful when the superword-mode is not activated: allows jumping to previous symbol while the word commands stop at each word separator character.
Move point forward to next target/ dependency	• **M-n** • **<f12> <down>** • **<M-f12> <down>** **<f11> SPC M <down>**	(**makefile-next-dependency**)	Move point to the beginning of the next dependency line. • Skips comments and macro definitions.
Move point backward to previous target/ dependency	• **M-p** • **<f12> <up>** • **<M-f12> <up>** **<f11> SPC M <up>**	(**makefile-previous-dependency**)	Move point to the beginning of the previous dependency line. • Skips comments and macro definitions.
Move point forward to next macro definition statement	• **<f12> <M-down>** • **<M-f12> <M-down>** **<f11> SPC M <M-down>**	(**pel-make-next-macro** &optional N SILENT DONT-PUSH-MARK	Move to the beginning of next N make file macro definition statement. • The function skips over comments. • If no valid form is found, don't move point, issue an error describing the failure unless SILENT is non-nil, in which case the function returns nil on error and non-nil on success. • The error message states the number of instanced searched, the regexp used and the number of instances found. • On success, the function push original position on the mark ring unless DONT-PUSH-MARK is non-nil. • The command support shift-marking.
Move point backward to previous macro definition statement	• **<f12> <M-up>** • **<M-f12> <M-up>** **<f11> SPC M <M-up>**	(**pel-make-previous-macro** &optional N SILENT DONT-PUSH-MARK)	Move to the beginning of previous N make file macro definition statement. • The function skips over comments. • If no valid form is found, don't move point, issue an error describing the failure unless SILENT is non-nil, in which case the function returns nil on error and non-nil on success. • The error message states the number of instanced searched, the regexp used and the number of instances found. • On success, the function push original position on the mark ring unless DONT-PUSH-MARK is non-nil. • The command support shift-marking.
iMenu/Speedbar See also: • ∑ **Completion/Input** • ∑ **Menus** • ∑ **Speedbar**	You can navigate through makefile macros and targets (identified as *dependencies*) using Emacs iMenu and Speedbar capabilities. • Several commands are available to get a list of the various elements and move point to it. • These commands include the following. More are listed in the ∑ **Completion/Input** . • Several packages extend the completion and how entry is done. PEL allows dynamic selection of several methods and can display the current status with **M-g ?** • You can also use the ∑ **Speedbar** to list all items on a vertical side-bar and navigate through them.		

Description	Keystroke	Function	Note
Find definitions using IMenu See also: • ∑ Completion/ Input • ∑ Menus	• `<f11> <f10> i` • `M-g i` • `M-g M-i`	(**imenu** INDEX-ITEM)	Lists imenu-detected items from the current buffer (according to its major mode). • For example, in a elisp file, the entry points are the function definitions and may include the variables and other items depending what function does the parsing (it can be semantic which provides more information). Provides one of the following interfaces to let user select entry to jump to: • The default: input completion, using the minibuffer window and tab completion. • a pop-up window : available in Graphics mode selected by mouse or in both graphics and terminal (TTY) modes when the **imenu-use-popup-menu** user-option is turned on. • with PEL you can use **pel-imenu-toggle-popup** (bound to `M-g <f4> p`) to toggle the user interface used by **imenu**.
Move to imenu detected symbol definition in current buffer ★★	• `M-g h` • `M-g M-h`	(**pel-goto-symbol**)	Prompt using for imenu symbol of the current buffer and move point to it. • Refresh imenu and jump to a place in the buffer using the completion method selected. • Modify user interface currently used with `M-g <f4> h`. • The command sets a ref-marker before moving. Return to previous location by typing `M-,`

Display current setting of commands:
• **pel-goto-symbol**
• **pel-goto-symbol-any-buffer**
See also:
• ∑ Completion/ Input
	`M-g ?`	(**pel-show-goto-symbol-settings**)	Display current settings used by the goto symbol commands in the echo area. For example:
			`goto-symbol UI is: popup-switcher`
			`goto-any-buffer UI is: Ido`
			`- iMenu lists are not flatten.`
			`- Ido uses:`
			`- Ido prompt geometry: grid mode, starts collapsed: expand with tab`
			`- Ido Ubiquitous mode: off`
			`- flx-ido mode: off`

Insert & Edit	The following commands help the editing of the makefile contents.		
Insert GNU make function statement	• `C-c Tab` • `C-c C-i`	(**makefile-insert-gmake-function**)	Insert a **GNU make function call**. • Asks for the name of the function to use (with completion). • Then prompts for all required parameters.
Insert target at point	`C-c :`	(**makefile-insert-target-ref** TARGET-NAME)	Complete on a list of known targets, then insert TARGET-NAME at point.
Add/remove line continuation trailing backslashes	`C-c C-\`	(**makefile-backslash-region** FROM TO DELETE-FLAG)	Insert, align, or delete end-of-line backslashes on the lines in the region. • With no argument, inserts backslashes and aligns existing backslashes. • With an argument, deletes the backslashes. This function does not modify the last line of the region if the region ends right at the start of the following line; it does not modify blank lines at the start of the region. So you can put the region around an entire macro definition and conveniently use this command.

Perform completion at point	`C-M-i`
	`<f12> .`
	`<f6> .`
		(**completion-at-point**)	Perform completion on the text around point.
			The completion method is determined by 'completion-at-point-functions'.
			⚠🚧 The `C-M-i` key sequence is also often bound to flyspell command.
			Use `<f12> .` instead.

Electric Insert	When the makefile-mode makefile-electric-keys user-option is turned on (it is off by default), the characters `$ : =` and `.` have special behaviour, described below.		
Insert macro reference	`$`	(**makefile-insert-macro-ref** MACRO-NAME)	Complete on a list of known macros, then insert complete ref at point.
Insert new target	`:`	(**makefile-electric-colon** ARG)	Prompt for name of new target. • Prompting only happens at beginning of line. • Anywhere else just self-inserts.
Insert macro defintion	`=`	(**makefile-electric-equal** ARG)	Prompt for name of a macro to insert. Only does prompting if point is at beginning of line. Anywhere else just self-inserts.
Insert special target	`.`	(**makefile-electric-dot** ARG)	Prompt for the name of a special target to insert. Supports tab completion. • Only does electric insertion at beginning of line. • Anywhere else just self-inserts.
Indenting	In make file editing, the tab character is important. The make program distinguish the tab character from multiple space characters. ⚠ The `C-M-q` key sequence is bound to prog-indent-sexp but it does not work well in makefile. Use the other 3 commands.		
Insert a tab character	`<tab>`	(**indent-for-tab-command** &optional ARG)	Inserts a tab character in a makefile.
Indent line(s) rigidly	• `<f6> <tab>` • `<f11> <tab> c`	(**pel-indent-lines** &optional N)	Indent current or marked lines by N indentation levels. Each level uses a tab character. • Works with point anywhere on the line. • All lines touched by the region are indented. • A special argument N can specify more than one indentation level. It defaults to 1. • If a negative number is specified, 'pel-unindent-lines' is used. • If a region is marked, the function does not deactivate it to allow repeated execution of the command. It also modifies the region to include all characters in all affected lines. • Use `C-g` to de-activate the region.
Un-indent line(s) rigidly	• `<backtab>` • `<f6> <backtab>` • `<f11> <tab> C`	(**pel-unindent-lines** &optional N)	• Un-indent current line or marked lines by N indentation levels. • Works with point is anywhere on the line. • All lines touched by the region are un-indented. • If region was marked, the function does not deactivate it to allow repeated execution of the command. • If a region was marked, the function does not deactivate it to allow repeated execution of the command. It also modifies the region to include all characters in all affected lines • Use `C-g` to de-activate the region.
Indent expression	`C-M-q`	(**prog-indent-sexp** &optional DEFUN)	Indent the expression after point. • When interactively called with prefix, indent the enclosing defun instead. ⚠ This command does not work well in makefiles.
Comment control	Although the make file modes provide the comment-region command, it's best to use comment-dwim as it works much better.		
Comment/un-comment See also:∑ Comments	`M-;`	(**comment-dwim** ARG)	Comment or un-comment line or region.

	• Comment or un-comment line or region.
	  • When no marked region and no comment:
	  • On empty line: insert comment starter at the proper indentation level. Typed again: move it toward end of line.
	  • On line with code: insert comment starter after the code for an end-of-line comment
	  • With marked un-commented region: Comment region (each line is commented)
	  • With marked commented region: Removes the comment.
	• Call the comment command you want (Do What I Mean).
	  • If the region is active and 'transient-mark-mode' is on, call 'comment-region' (unless it only consists of comments, in which case it calls 'uncomment-region'). Else, if the current line is empty, call 'comment-insert-comment-function' if it is defined, otherwise insert a comment and indent it. Else if a prefix ARG is specified, call 'comment-kill'. Else, call 'comment-indent'.		

	`C-c C-c`	(**comment-region** BEG END &optional ARG)	Comment or uncomment each line in the region. ⚠ Prefer **comment-dwim**: it works better.

	Comment or uncomment each line in the region.
	  • With just C-u prefix arg, uncomment each line in region BEG .. END.
	  • Numeric prefix ARG means use ARG comment characters. If ARG is negative, delete that many comment characters instead.
	• The strings used as comment starts are built from 'comment-start' and 'comment-padding'; the strings used as comment ends are built from 'comment-end' and 'comment-padding'.
	• By default, the 'comment-start' markers are inserted at the current indentation of the region, and comments are terminated on each line (even for syntaxes in which newline does not end the comment and blank lines do not get comments). This can be changed with 'comment-style'.		

Analyze	The following commands analyze the content of the make file or the file system.		
Scan current directory files, checking for targets	`C-c C-f`	(**makefile-pickup-filenames-as-targets**)	Scan the current directory for filenames to use as targets. • Checks each filename against 'makefile-ignored-files-in-pickup-regex' and adds all qualifying names to the list of known targets.

Description	Keystroke	Function	Note
Scan current buffer for makefile content	`C-c C-p`	(**makefile-pickup-everything** ARG)	Notice names of all macros and targets in Makefile. • Prefix arg means force pickups to be redone. Use this to refresh the list of macros and targets located in the makefile before executing another action on those.
Update scan with latest makefile buffer content	`C-c C-u`	(**makefile-create-up-to-date-overview**)	Create a buffer containing an overview of the state of all known targets. • Known targets are targets that are explicitly defined in that makefile; in other words, all targets that appear on the left hand side of a dependency in the makefile.
List macros and targets in dedicated buffer	`C-c C-b`	(**makefile-switch-to-browser**)	Open a *Macros and Target* buffer that only lists them. • It operates in Fundamental mode and aside listing the macros and targets provides nothing more.

Emacs & Makefile— References

Document	Notes
Make tools	See also: **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**
Makepp home page	Makepp, also called make++ is a GNU Make replacement, written in Perl. It addresses the recursive make problem.
Make generic information	
Recursive Make Considered Harmful - Steve Miller	PDF paper (from the wayback machine archive) written by Steve Miller in 1997 describing the concept of recursive make technique showing why it causes several problems and what can be done to avoid them.
Non-Recursive Make Considered Harmful	A march 2016 PDF paper from Andrey Mokhov, Neil Mitchell, Simon Peyton Jones and Simon Marlow describe how even a non-recursive make based build system can be difficult to maintain and they propose something based on the Shake Haskell library.

GNU Make Rules

GNU Make Rules

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	**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	**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 patter or all): • **vpath *pattern directories*** • **vpath *pattern*** • **vpath**	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 phone 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`
	• 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

Topic			
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 therm 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.

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.	
	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)`

OP	Description	Example

:=	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`

?=	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`