↖ Topic Index

Emacs support for Unix Shell Scripting

Description	Keystroke	Function	Note

UNIX-like Shell Script Editing
See:
• **comparison of command shells**
• **ShellCheck Wiki**
• **ShellCheck on-line**
 	
• **PEL sh support activation** ☛
Activate sh-mode on files ☛
 	
• Make script executable ☛
• Distinguish script from sourced scripts ☛
• Script extensions ☛
• Σ **Indentation** control ☛
• shellcheck syntax check ☛
 	
• Specialized templates ☛
• superword-mode on ☛
	Emacs provides the built-in **sh-mode** to support UNIX-style shell script programming.
	• It supports several shell variants including:
	  • **bash** - see **Bash Reference Manual**
	  • **csh** - see **An Introduction to C shell** , **csh OpenBSD man page**, **csh NetBSD Man page**,
	  • **ksh**,
	  • **sh**, the Bourne shell
	  • **zsh** - see **zsh Manual** and **The Z Shell** page
	  Several other shell types are supported . Use the **sh-set-shell** command to force the use of a specific shell type, with **C-c :**
	PEL activates Unix shell-script support with the **pel-use-sh** user-options.
	  • When **pel-use-sh** on: the **<f11> SPC H** prefix is made available. In a shell script buffer these commands are accessible via the **<f12>** key.
	The **auto-mode-alist** user-option identifies path patterns files that must use the **sh-mode** or **shell-script-mode** (which is an alias for sh-mode).
	• **pel-auto-mode-alist**: identifies extra entries that PEL automatically adds to the auto-mode-alist.
	  • Add `/bin/[^.]+\'` to sh-mode to automatically activate sh-mode for your shell scripts stored inside your ~/bin directory.
	• PEL also activate extra minor modes in shell-script-mode through the PEL **pel-sh-activates-minor-modes** user-option.
	
	• **pel-make-script-executable** : when turned on (set to t), Emacs makes the saved shell script file executable.
	• PEL provides the ability to automatically identify shell scripts that must be sourced and are therefore not executables:
	  • **pel-shell-sourced-script-file-name-prefix**: use a regexp to identify the base name of files that are meant to be sourced. For example, if all shell files that are sourced have a file name that begins with an underscore, use the following regexp: **\`_**
	  • **pel-shell-script-extensions**: identifies file extensions of files that PEL must **not** identify as sourced files.
	• Use of hard tab for indentation is set by **pel-sh-use-tabs**. The number of columns used for indentation is controlled by **pel-sh-tab-width**.
	• Set **pel-use-shellcheck** to activate shellcheck-based syntax checking. Values allow activating flycheck or flymake manually or automatically. Recommendation: select 'use flycheck automatically': it will activate it and will provide key bindings automatically.
	• PEL also provide specialized code templates that are taking the above user-options into account. The commands distinguish a shell script file that must be executable from one that must be sourced and generates different text.
	• PEL activates the **superword-mode** automatically in shell script buffers. See Σ **Text Modes** for more info.		

Open this PDF file. See also: Σ **Help/Info**	**<f11> SPC H <f1**> **<f12> <f1>**	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **PEL - UNIX Shell** 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.
Σ **Customize** PEL UNIX Shell support	**<f11> SPC H <f2>** **<f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL UNIX Shell support. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Σ **Customize** Emacs UNIX Shell support	**<f11> SPC H <f3>** **<f12> <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs UNIX Shell support: sh, sh-script, sh-indentation. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Specialized Execution	The following commands can be used to change the scripting dialect and to execute a portion of the code in the buffer.		
Set the buffer shell type	**C-c :**	(**sh-set-shell** SHELL &optional NO-QUERY-FLAG INSERT-FLAG)	Set this buffer's shell to SHELL (a string). Prompts, support tab-completion. • When used interactively, insert the proper starting #!-line, and make the visited file executable via 'executable-set-magic', perhaps querying depending on the value of 'executable-query'. • Calls the value of 'sh-set-shell-hook' if set. • Shell script files can cause this function be called automatically when the file is visited by having a 'sh-shell' file-local variable whose value is the shell name (don't quote it).
Execute region in a sub-shell	**C-M-x**	(**sh-execute-region** START END &optional FLAG)	Pass optional header and region to a subshell for noninteractive execution. • The working directory is that of the buffer, and only environment variables are already set which is why you can mark a header within the script. • With a positive prefix ARG, instead of sending region, define header from beginning of buffer to point. With a negative prefix ARG, instead of sending region, clear header. • Print result on the echo area if it fits, otherwise into the *Shell Command Output* buffer.
Syntax checking with shellcheck	Emacs shell script buffer syntax checking is done by **shellcheck**. It can be provided by the built-in **flymake** or the **flycheck** external package. With PEL, the **pel-use-shellcheck** user-option determines which one is supported, if any. Defaults to no support.		

Flycheck
 	
pel-use-shellcheck :=
• flycheck-manual
• flycheck-automatic
	Flycheck is a minor mode for on-the-fly syntax checking.
	The **flycheck** external package is activated by PEL when **pel-use-shellcheck** is set to either flycheck-manual or flycheck-automatic.
	  • It is also activated when the **pel-use-flycheck** user-option is turned on when another major mode specific user-option requires it.
	Aside from the following 2 key bindings that PEL provides to toggle the flycheck mode,
	  flycheck key prefix is **C-c !** as set by its **flycheck-keymap-prefix** user-option. You can change it for a different key prefix.		

Toggle flycheck mode for current buffer	**<f11> ! !**	(**flycheck-mode** &optional ARG)	Toggle flycheck minor-mode for the current buffer.
Toggle flycheck mode for all buffers	**<f11> ! M-!**	(**global-flycheck-mode** &optional ARG)	Toggle Flycheck mode in all buffers. • Flycheck mode is enabled in all buffers where 'flycheck-mode-on-safe' would do it.
• **Info about Flycheck**	The following extra key bindings are available when flycheck is active.		
Open Flycheck manual	**C-c ! i**	(**flycheck-manual**)	Open the Flycheck manual.
Display Flycheck version	**C-c ! V**	(**flycheck-version** &optional SHOW-VERSION)	Get the Flycheck version as string. • If called interactively or if SHOW-VERSION is non-nil, show the version in the echo area and the messages buffer. • The returned string includes both, the version from package.el and the library version, if both a present and different. • If the version number could not be determined, signal an error, if called interactively, or if SHOW-VERSION is non-nil, otherwise just return nil.
• **Flycheck setup**	The following extra key bindings are available when flycheck is active.		
Display documentation about syntax checker	**C-c ! ?**	(**flycheck-describe-checker** CHECKER)	Display the documentation of CHECKER. • CHECKER is a checker symbol. • Pop up a help buffer with the documentation of CHECKER.
Select Flycheck Checker for current buffer	**C-c ! s**	(**flycheck-select-checker** CHECKER)	Select CHECKER for the current buffer. • CHECKER is a syntax checker symbol (see 'flycheck-checkers') or nil. In the former case, use CHECKER for the current buffer, otherwise deselect the current syntax checker (if any) and use automatic checker selection via 'flycheck-checkers'. • If called interactively prompt for CHECKER. With prefix arg deselect the current syntax checker and enable automatic selection again. • Set 'flycheck-checker' to CHECKER and automatically start a new syntax check if the syntax checker changed. • CHECKER will be used, even if it is not contained in 'flycheck-checkers', or if it is disabled via 'flycheck-disabled-checkers'.
Verify Flycheck setup	**C-c ! v**	(**flycheck-verify-setup**)	Check whether Flycheck can be used in this buffer. • Display a new buffer listing all syntax checkers that could be applicable in the current buffer. For each syntax checkers, possible problems are shown.
Disable Flycheck checker	**C-c ! x**	(**flycheck-disable-checker** CHECKER &optional ENABLE)	Interactively disable CHECKER for the current buffer. • Prompt for a syntax checker to disable, and add the syntax checker to the buffer-local value of 'flycheck-disabled-checkers'. • With non-nil ENABLE or with prefix arg, prompt for a disabled syntax checker and re-enable it by removing it from the buffer-local value of 'flycheck-disabled-checkers'.

Description	Keystroke	Function	Note
• **Flycheck buffer/file**	The following extra key bindings are available when flycheck is active.		
Syntax Check current buffer	`C-c ! c`	(**flycheck-buffer**)	Start checking syntax in the current buffer. • Get a syntax checker for the current buffer with 'flycheck-get-checker-for-buffer', and start it.
Check syntax of current file	`C-c ! C-c`	(**flycheck-compile** CHECKER)	Run CHECKER via 'compile'. • Prompt for a syntax checker to run. • Instead of highlighting errors in the buffer, this command pops up a separate buffer with the entire output of the syntax checker tool, just like 'compile'.
• **Manage Errors**	The following extra key bindings are available when flycheck is active.		
Show error list for current buffer	• `C-c ! l` • `<f12> e`	(**flycheck-list-errors**)	Show the error list for the current buffer.
Display all errors at point	`C-c ! h`	(**flycheck-display-error-at-point**)	Display all the error messages at point.
Explain error at point	• `C-c ! e` • `<f12> /`	(**flycheck-explain-error-at-point**)	Display an explanation for the first explainable error at point. • In a shell script buffer this opens the **shellcheck wiki page** for the identified error.
Copy errors	`C-c ! C-w`	(**flycheck-copy-errors-as-kill** POS &optional FORMATTER)	Copy each error at POS into kill ring, using FORMATTER. • FORMATTER is a function to turn an error into a string, defaulting to 'flycheck-error-message'. • Interactively, use 'flycheck-error-format-message-and-id' as FORMATTER with universal prefix arg, and 'flycheck-error-id' with normal prefix arg, i.e. copy the message and the ID with universal prefix arg, and only the id with normal prefix arg.
Clear all errors	`C-c ! C`	(**flycheck-clear** &optional SHALL-INTERRUPT)	Clear all errors in the current buffer. • With prefix arg or SHALL-INTERRUPT non-nil, also interrupt the current syntax check.
Move point to next error	• `C-c ! n` • `M-n`	(**flycheck-next-error** &optional N RESET)	Visit the N-th error from the current point. • N is the number of errors to advance by, where a negative N advances backwards. With non-nil RESET, advance from the beginning of the buffer, otherwise advance from the current position.
Move point to prior error	• `C-c ! p` • `M-p`	(**flycheck-previous-error** &optional N)	Visit the N-th previous error. • If given, N specifies the number of errors to move backwards by. • If N is negative, move forwards instead.
Specialized Navigation	The following commands override normal key bindings and provide specialized navigation key bindings in shell scripts buffers.		
Go to beginning of command	`M-a`	(**sh-beginning-of-command**)	Move point to successive beginnings of commands.
Go to end of command	`M-e`	(**sh-end-of-command**)	Move point to successive ends of commands.
Backward to beginning of block: • **if*** ⇐ • **for | while | until** ⇐ • **case** ⇐ **(block backward)** See also: **Navigation**	• `C-M-b` • `C-M-<left>` • `C-[ C-b` • `Esc C-b` • `Esc C-<left>`	(**backward-sexp** &optional ARG)	Move backward across one balanced expression (sexp). • With ARG, do it that many times. Negative arg -N means move forward across N balanced expressions. This command assumes point is not in a string or comment. • `C-M-b` : ☛ Shift marking is available in graphics mode, not in terminal mode. • `C-M-<left>` : ☛ Shift marking works with this command. • ⚠ With PEL: if you want to use `Esc C-<left>` binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. ❖ `C-M-<left>` does not work on Windows, but `H-<left>` works. Several Linux distros map `C-M-<left>` to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.
Forward to end of block: • ⇒ **fi** • ⇒ **done** • ⇒ **esac** **(block forward)** See also: **Navigation**	• `C-M-f` • `C-M-<right>` • `C-[ C-f` • `Esc C-f` • `Esc C-<right>`	(**forward-sexp** &optional ARG)	Move forward across one balanced expression (sexp). • With ARG, do it that many times. Negative arg -N means move backward across N balanced expressions. This command assumes point is not in a string or comment. • `C-M-f` : ☛ Shift marking is available in graphics mode, not in terminal mode. • `C-M-<right>` : ☛ Shift marking works with this command. • ⚠ With PEL: if you want to use `Esc C-<right>` binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. ❖ `C-M-<right>` does not work on Windows, but `H-<right>` does. Several Linux distros map `C-M-<right>` to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.
Using Flymake **pel-use-shellcheck** := • flymake-manual • flymake-automatic	You can also use Emacs built-in flymake to control shell-check based syntax checking. ⚠ Note, however, than using flymake does not provide as many commands as when you use flycheck (as described above). • Several key bindings are not available when flymake is used. Flymake has several customizable variables, which some listed here: The following customization variables determine the exact circumstances whereupon Flymake decides to initiate a check of the buffer: • **flymake-start-on-flymake-mode** : **t** to start checking when flymake-mode is started. **nil** to prevent check. • **flymake-no-changes-timeout** : time to wait after last change to start checking. Default = 0.5 seconds. • **flymake-start-syntax-check-on-newline** : **t** to check after insertion or removal of newline char from buffer. **nil** to prevent check. The following variable control navigation to next or previous error: • **flymake-wrap-around** : If non-nil, moving to errors wraps around buffer boundaries. • **flymake-diagnostic-types-alist** : Alist ((KEY . PROPS)*) of properties of Flymake diagnostic types. See Emacs documentation for more info.		
Toggle Flymake mode on/off	`M-x flymake-mode`	(**flymake-mode** &optional ARG)	Toggle Flymake mode on or off. • With a prefix argument ARG, enable Flymake mode if ARG is positive, and disable it otherwise. • Flymake is an Emacs minor mode for on-the-fly syntax checking. • Flymake collects diagnostic information from multiple sources, called backends, and visually annotates the buffer with the results.
Go to next flymake diagnostic	`M-n`	(**flymake-goto-next-error** &optional N FILTER INTERACTIVE)	Move point to the next Flymake diagnostic. • With a prefix arg, skip any diagnostics with a severity less than ':warning'. • Display the error message in the echo line.
Go to previous flymake diagnostic	`M-p`	(**flymake-goto-prev-error** &optional N FILTER INTERACTIVE)	Move point to the previous Flymake diagnostic. • With a prefix arg, skip any diagnostics with a severity less than ':warning'. • Display the error message in the echo line.

Description	Keystroke	Function	Note
Comments	Insert a comment, comment or un-comment a region with `M-;`		
Toggle display of comments in buffer or active region See also: **Comments**	`<f11> ; ;`	(**hide/show-comments-toggle** &optional START END)	Toggle hiding/showing of comments in the active region or whole buffer. • If the region is active then toggle in the region. Otherwise, in the whole buffer. 📦 This requires the **hide-comnt.el** package (see **Comments**). PEL activates it when the **pel-use-hide-comnt** user option is **t**.
Specialized Insertion			
Double quote word at point	`<f12> "`	(**pel-sh-double-quote-word**)	Surround word at point or selected area with double quotes.
Singe quote word at point	`<f12> '`	(**pel-sh-single-quote-word**)	Surround word at point or selected area with single quotes.
Backtickquote word at point	`<f12> ` `	(**pel-sh-backtick-quote-word**)	Surround word at point or selected area with back-tick characters.
Generic code skeletons • **tempo skeletons** See also: • **Inserting Text** • **T Templates**	Several mechanisms have been developed to allow easy insertion of predefined text in Emacs. • Emacs provides the built-in skeleton mechanism and the **tempo skeletons**. • PEL supports both. They are used a little bit differently. • PEL provides key bindings to the tempo skeletons: the generic code templates, accessible via the **<f6>** prefix key, and the language-specific code templates, accessible via the **<f12>** key prefix. PEL provides **generic** tempo skeletons the handle UNIX shell script files.		
Customize PEL Text Insertions control	`<f6> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL generic tempo skeleton customization groups that control the format of the various skeletons including the generic skeleton used by the `<f6> h` key (se below). • If OTHER-WINDOW is non-nil (use `C-u`), display in other window.
Insert generic file module header block — Language agnostic After inserting the template, navigate though areas that must be filled with: • tempo-forward-mark: **C-c .** • tempo-backward-mark: **C-c ,**	`<f6> h`	(**pel-generic-file-header**)	Insert a file header block at the top of the file. Works only for buffer visiting a file. ⚠️ The command key binding `<f6> h` is available only 1 second after Emacs has started. ☝Specify the format of the header via the user-options in the **pel-pkg-generic-code-style** customization group accessible via `<f6> <f2>` • Inside a **sh-mode** buffer, `<f12> <f2>` provides access to the following customization groups: • **pel-pkg-for-sh** for the control of the template format and **pel-sh-script-skeleton-control** for sh-mode specific user-options. • The files that have no extensions are often used in Unix-like OS shell scripts. • These files are also supported as Emacs can recognize them if they are stored in a **bin** directory. ☝After inserting a template, use **tempo-forward-mark** and **tempo-backward-mark** to move to the beginning of each section that must be filled.
Toggle pel-tempo-mode	`<f6> SPC`	(**pel-tempo-mode** &optional ARG)	Toggle PEL tempo mode on/off. PEL tempo mode activates `C-c .` and `C-c ,` as well as to `C-c C-.` and `C-c C-,` key bindings to navigate across tempo mark hot-spots. When pel-tempo-mode is active the pel-tempo-mode lighter (‡) is shown on the status bar. The second set of keys are only available when Emacs runs in graphics mode. ☝The pel-generic-file-header command inserts the text using a tempo skeleton: the PEL tempo mode is automatically activated by typing `<f6> h`.
Jump to next tempo mark	• `C-c M-f` • `C-c .` • `C-c C-.`	(**tempo-forward-mark**)	Jump to the next mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key key bindings are only available when pel-tempo-mode is active.
Jump to previous tempo mark	• `C-c M-b` • `C-c ,` • `C-c C-,`	(**tempo-backward-mark**)	Jump to the previous mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key binding are only available when pel-tempo-mode is active.
Shell statement Insertion	The sh-mode provides the following commands to insert shell scripts code elements with templates defined with the **Emacs skeleton language**. All of these statement insertion command share the same extra description: • This is a skeleton command (see 'skeleton-insert'). • Normally the skeleton text is inserted at point, with nothing "inside". • If there is a highlighted region, the skeleton text is wrapped around the region text. • A prefix argument ARG says to wrap the skeleton around the next ARG words. • A prefix argument of -1 says to wrap around region, even if not highlighted. • A prefix argument of zero says to wrap around zero words---that is, nothing. • This is a way of overriding the use of a highlighted region.		
Insert a case/switch	`C-c C-c`	(**sh-case** &optional STR ARG)	Insert a case/switch statement.
Insert a for loop	`C-c C-f`	(**sh-for** &optional STR ARG)	Insert a for loop.
Insert function definition	`C-c (`	(**sh-function** &optional STR ARG)	Insert a function definition.
Insert a if statement	• `C-c <tab>` • `C-c C-i`	(**sh-if** &optional STR ARG)	Insert a if statement.
Insert an indexed loop from 1 to n.	`C-c C-l`	(**sh-indexed-loop** &optional STR ARG)	Insert an indexed loop from 1 to n.
Insert a getopt loop	`C-c C-o`	(**sh-while-getopts** &optional STR ARG)	Insert a while getopts loop. • Prompts for an options string which consists of letters for each recognized option followed by a colon ':' if the option accepts an argument.
Insert a repeat loop definition	`C-c C-r`	(**sh-repeat** &optional STR ARG)	Insert a repeat loop definition.
Insert a select statement	`C-c C-s`	(**sh-select** &optional STR ARG)	Insert a select statement.
Insert an until loop	`C-c C-u`	(**sh-until** &optional STR ARG)	Insert an until loop.
Insert a while loop	`C-c C-w`	(**sh-while** &optional STR ARG)	Insert a while loop.
Show indentation	`C-c ?`	(**sh-show-indent** ARG)	Show how the current line would be indented. • This tells you which variable, if any, controls the indentation of this line. • If optional arg ARG is non-null (called interactively with a prefix), a pop up window describes this variable. • If variable 'sh-blink' is non-nil then momentarily go to the line we are indenting relative to, if applicable.
Set indentation for current line	`C-c =`	(**sh-set-indent**)	Set the indentation for the current line. If the current line is controlled by an indentation variable, prompt for a new value for it.
Learn indentation from current line	`C-c <`	(**sh-learn-line-indent** ARG)	Learn how to indent a line as it currently is indented. • If there is an indentation variable which controls this line's indentation, then set it to a value which would indent the line the way it presently is. • If the value can be represented by one of the symbols then do so unless optional argument ARG (the prefix when interactive) is non-nil.

Description	Keystroke	Function	Note
Learn indentation from buffer	`C-c >`	(**sh-learn-buffer-indent** &optional ARG)	Learn how to indent the buffer the way it currently is. • If ‘sh-use-smie’ is non-nil, call ‘smie-config-guess’. Otherwise, run the sh-script specific indent learning command, as described below. • Output in buffer "*indent*" shows any lines which have conflicting values of a variable, and the final value of all variables learned. • When called interactively, pop to this buffer automatically if there are any discrepancies. • If no prefix ARG is given, then variables are set to numbers. • If a prefix arg is given, then variables are set to symbols when applicable -- e.g. to symbol ‘+’ if the value is that of the basic indent. • If a positive numerical prefix is given, then ‘sh-basic-offset’ is set to the prefix’s numerical value. • Otherwise, sh-basic-offset may or may not be changed, according to the value of variable ‘sh-learn-basic-offset’. • Abnormal hook ‘sh-learned-buffer-hook’ if non-nil is called when the function completes. The function is abnormal because it is called with an alist of variables learned. ⚠ This command can often take a long time to run.

Zsh — The Z Shell

Zsh - Configuration

File	Users	Configuration file sourced order (See note 2)					Sourced for zsh -f	Similar to	Notes
		New Terminal session: Interactive Login	**Sub-shell:** Interactive non-login **(**includes Emacs shells**)**	**Scripts :** non-interactive, non-login	**Scripts with special flag or piping output into SSH connection:** Non-interactive login	**Logout**			Read section STARTUP/SHUTDOWN Files in the **zsh(1)** man page for information on configuration files. Use **Emacs** to display your local zsh man pages. See **Help/Info** .
/etc/zshenv	All	**1**	**1**	**1**	**1**	Not sourced	**Yes**		
$ZDOTDIR/.zshenv	Current	**2**	**2**	**2**	**2**	Not sourced	No		• If ZDOTDIR undef, uses $HOME • Do **NOT** set PATH here!
/etc/zprofile	All	**3**	Not sourced	Not sourced	**3**	Not sourced	No	~/.profile of: • **bash** • **sh**	⚠️ On macOS, /etc/zprofile eval /usr/libexec/path_helper to set PATH.
$ZDOTDIR/.zprofile	Current	**4**	Not sourced	Not sourced	**4**	Not sourced	No		• If ZDOTDIR undef, uses $HOME • Define EDITOR, prepend|append user PATH **here!**
/etc/zshrc	All	**5**	**3**	Not sourced	Not sourced	Not sourced	No		
$ZDOTDIR/.zshrc	Current	**6**	**4**	Not sourced	Not sourced	Not sourced	No		• If ZDOTDIR undef, uses $HOME • Define functions, aliases, prompt **here!**
/etc/zlogin	All	**7**	Not sourced	Not sourced	**5**	Not sourced	No	~/.login of: • **ksh**	
$ZDOTDIR/.zlogin	Current	**8**	Not sourced	Not sourced	**6**	Not sourced	No		• If ZDOTDIR undef, uses $HOME
/etc/zlogout	All	Not sourced	Not sourced	Not sourced	Not sourced	**1**	No		
$ZDOTDIR/.zlogout	Current	Not sourced	Not sourced	Not sourced	Not sourced	**2**	No		• If ZDOTDIR undef, uses $HOME

Zsh - Notes

#	Topic	Note
1	**ZDOTDIR**	The location of the user specific configuration files can be modified by ZDOTDIR environment variable. • ☝Instead of using ZDOTDIR envar you can also place symlinks to the real location of the files.
2	**Configuration Files**	
	Login	When opening a new terminal window.
	Interactive shell	When executing **zsh** explicitly (or implicitly inside an editor like Emacs).
	Script	When executing a script that has a #!/bin/zsh shebang.
	Logout	Used as cleanup of login shell, executed before terminating the process of a login shell. If the shell is terminated by an external process these files might not run.
3	**Envvar**	Environment variables built-in shells that are quite useful.
	$	Print the current process ID with: `echo "$$"` . The process ID of the shell parent process is $**PPID**
	0	Name of the shell (really the name of the program that launched the shell). Print it with: `echo "$0"`
	?	Exit code value of the last executed command. Print it with: `echo "$?"` .
	SHLVL	An environment variable that represents the depth of a stack of shells executing each other. Supported by zsh, but also bash, ksh

Zsh - References

Topic	Reference
About Zsh	• **Z shell @ Wikipedia**
Zsh Home	• **Zsh home page** : with links to the following: • **Z shell page**. : with links to information, documentation, support and source code pages.

Zsh intro, manual & tips	• The zsh Manual:
	  • **Zsh Manual** (the **manual in several formats**) : the official zsh manual — current version.
	  • **Introduction to Zsh** (a November 30, 1995 document; gives an overview of Zsh early on).
	  • **A User's Guide to the Z-Shell**, by Peter Stephenson March 23, 2003.
	  • This also is a predecessor to the book **From Bash to Z Shell** published by APress in 2005
	• **Moving to Zsh**, from Armin Briegel, provides a good introduction to macOS users.
	• **Awesome-zsh** lists several things zsh can do in an easy-to-read list format.

Zsh configuration info	• **How do Zsh Configuration Files Work? By Daniel Kehoe @ freeCodeCamp .** Good into but 100% correct as it misses identifying some files. • **Zsh prompt expansion** which lists the codes used to customize zsh prompt. • **Customizing the zsh Prompt** , from Armin Briegel
Zsh configuration frameworks	• **Oh My Zsh** - An open-source, community-driven framework for managing Zsh configuration. • **Pretzo** • **Zimfw** • **Zsh-snap**

Other Zsh resources	• **Z-Shell.dev,** a wiki-based collection of tools for zsh. Documentation is very scarce and would benefit from a overview description of the site's purpose. It seems to be devoted to **Zi**, but the purpose of Zi is not clear either. Their code is not well documented, an unfortunate trend in software these days. However some potentially useful pages might be useful:
	  • **Zsh Plugin Standard** — seems to list a set of conventions - at this point I'm not sure if these are followed globally or it's just for **Zi**.
	  • **Zsh Native Scripting Handbook** — a list of Zsh script snippets.

Zsh man pages	☝Use **Emacs** to display zsh man pages! Viewing man pages with Emacs gives you great navigation and control. • See **Help/Info** toward end of page 5 for the commands to open a man page. Use `M-<f8>` with PEL.
Zsh Linters	High-quality linter for Zsh do not seem to exist. **Shellcheck** mostly supports bash and sh, not zsh specifics. The other two Zsh linters I found seem to be in early or abandoned state: **zlint** and **zsh-lint**.
Other Topics	• **Reload Zsh configuration** - Bozhidar Batsov article