CRiSP/CRiSPer to Emacs/PEL conversion

Concept	CRiSP key	Emacs Key	command	Description
CRiSP and CRiSPer	This reference table lists correspondence between some of the commands available in: • the CRiSP editor using the CRiSPer extensions I (Pierre Rouleau) developed for myself in the past and, • Emacs with PEL. **CRiSP** is a nice commercial editor written by Paul Fox that was derived from once very popular **Underware's Brief** editor. CRiSP is still available today and it is a fine editor for lots of purposes despite its age (but Emacs is older and I suspect that the Brief designers were aware of Emacs in the 80's because Brief underlying macro language was a simple Lisp). Emacs audience is wider and it is nicely maintained. I no longer maintain the CRiSPer extensions and I never released them as open source or otherwise. I am instead concentrating on Emacs today. Emacs has similar features and I have implemented some of the CRiSPer features (but not all) inside PEL but mostly differently with extensions of the features to take advantage of what Emacs offers. This table contains a partial a list of the CRiSP/CRiSPer commands and their Emacs equivalent.			
Resize window See: ⟫ **Windows**	**<f2>** • resize window in direction of arrow key	Emacs does not have a clean window resize like CriSP. You can resize the windows with the mouse and use commands to increase or decrease the size of the current window vertically and horizontally. Therefore in Emacs several commands are required, listed below. • The **<f7>** keys are part of the PEL Window Hydra: you type **<f7>** and one of the keys (like v), then you can type the other Hydra keys without typing <f7>. You type the <f7> key to stop using the Hydra and return to normal typing mode. • 📦 The Hydra keys requires the **hydra** external package. 🔄 With PEL user option **pel-use-hydra** set to **t**, PEL activates the **hydra external package** and also creates a Hydra set of keys to help speed up navigation and management of windows. See: ⟫ **Windows** for more info.		
	Grow window taller	• **C-x ^** • **<f11> w s V** • **ESC M-<up>** • **<f1> M-<up>** • **<f7> V**	(**enlarge-window** DELTA &optional HORIZONTAL)	Grow window taller by DELTA lines (defaults to 1), specify more with **C-u** n (or **M-** n) argument prefix. • See note above for availability of various bindings.
	Shrink window smaller	• **<f11> w s v** • **ESC M-<down>** • **<f1> M-<down>** • **<f7> v**	(**shrink-window** DELTA &optional HORIZONTAL)	Shrink height of window by DELTA lines (defaults to 1), specify more with **C-u** n (or **M-** n) argument prefix. • See note above for availability of various bindings.
	Grow windows wider	• **C-x }** • **<f11> w s H** • **ESC M-<right>** • **<f1> M-<right>** • **<f7> H**	(**enlarge-window-horizontally** DELTA)	Enlarge the current window horizontally. • See note above for availability of various bindings.
	Shrink window narrower	• **C-x {** • **<f11> w s h** • **ESC M-<left>** • **<f1> M-<left>** • **<f7> h**	(**shrink-window-horizontally** DELTA)	Reduce the width of the current window. • See note above for availability of various bindings.
	Make all windows the same size	• **C-x +** • **<f11> w s =** • **ESC <kp-5>** • **<f1> <kp-5>** • **<f7> =**	(**balance-windows** &optional WINDOW-OR-FRAME)	Balance the sizes of windows of WINDOW-OR-FRAME. • WINDOW-OR-FRAME is optional and defaults to the selected frame. • If WINDOW-OR-FRAME denotes a frame, balance the sizes of all windows of that frame. If WINDOW-OR-FRAME denotes a window, recursively balance the sizes of all child windows of that window. • See note above for availability of various bindings.

Split window
 	
See: ⟫ **Windows**
	<f3>
	• split window pointed to by arrow key
		• Emacs native commands are **C-x 2** and **C-x 3** to split window of 2 windows on top of each other (C-x 2) or side by side (C-x 3).
		• I added several keys:
		  • in the PEL Window Hydra (the keys that are listed as beginning with <f7>)
		  • and the other keys that start with the ESC, <f1> or <f11> prefix.		

	Create new window below	• **C-x 2** • **<f7> 2** • **<f7> -**	(**split-window-below** &optional SIZE)	Split the selected window into two windows, one above the other. • The selected window is above. The newly split-off window is below and displays the same buffer. ☛ Note that Emacs default behaviour attempts to maximize the view into the current buffer when splitting the buffer into 2 windows. This means that the cursor will not be located in the same position in the new window. To change this behaviour and keep the same point in both windows, execute *(setq split-window-keep-point nil).* The PEL packages does that.
	Create new window at right	• **C-x 3** • **<f7> 3** • **<f7> |**	(**split-window-right** &optional SIZE)	Split the selected window into two side-by-side windows. • The selected window is on the left. The newly split-off window is on the right and displays the same buffer.

	Create window at cursor direction	• **ESC C-<right>**
		• **ESC C-<left>**
		• **ESC C-<down>**
		• **ESC C-<up>**
		• **<f1> C-<right>**
		• **<f1> C-<left>**
		• **<f1> C-<down>**
		• **<f1> C-<up>**
		• **<f11> C-<right>**
		• **<f11> C-<left>**
		• **<f11> C-<down>**
		• **<f11> C-<up>**
		• **<f7> C-<right>**
		• **<f7> C-<left>**
		• **<f7> C-<down>**
		• **<f7> C-<up>**
			• **(pel-create-window-right)**
			• **(pel-create-window-left)**
			• **(pel-create-window-down)**
			• **(pel-create-window-up)**
				Create a window at the location pointed by the cursor's direction, and move point inside the new window.
				
				• The 4 different commands and shown in the same cell for convenience, one for each of the available cursors: **<right>**, **<left>**, **<down>** and **<up>**.
				• There are 4 possible sets of bindings:
				  • 3 sets of stand-alone commands:
				  • Commands with **<f11>** prefix, always available.
				  • Commands with **ESC** prefix, 🔄 available when **pel-windmove-on-esc-cursor** user option is on (set to t).
				  • Commands with **<f1>** prefix, 🔄 available when **pel-windmove-on-f1-cursor** user option is on (set to t).
				  • The Hydra-based commands, with the Hydra activated with any of the key sequences that use the **<f7>** prefix. 🔄 Available when **pel-use-hydra** user option is set to t.

Close Window See: ⟫ **Windows**	**<f4>** • kill window pointed to be arrow key	• Emacs provides several ways to close windows. When a Emacs window is closed, the buffer is normally not killed (and therefore if it contained a file, nothing happens to the file) . Some commands allow killing the buffer at the same time as closing the window though. • Some of the commands below (the one that start with <f7>) are part of the PEL Window Hydra.		
	Close this windows	• **C-x 0** • **<f7> 0** • **<f7> d**	(**delete-window** &optional WINDOW)	This just closes the window and moves the cursor to the next window.
	Kill current buffer and close window See also: ⟫ **Buffers**	• **C-x 4 0** • **<f7> k**	(**kill-buffer-and-window**)	Kill the current buffer and delete the selected window.

Concept	CRiSP key	Emacs Key	command	Description
	Close a window identified by number	**<f11> w k**	(**ace-delete-window**)	Delete a window selected by a number, a number shown in the top-left corner of the window. 📦 Requires the **ace-window** external package. PEL downloads, installs and activates it when the **pel-use-ace-window** user options is set to **t**.
	Close all other windows	• **C-x 1** • **<f7> 1** • **<f7> .**	(**delete-other-windows** &optional WINDOW)	Make current window fill its frame.

	Close a window at cursor direction	• **ESC C-S-<right>**
		• **ESC C-S-<left>**
		• **ESC C-S-<down>**
		• **ESC C-S-<up>**
		
		• **<f1> C-S-<right>**
		• **<f1> C-S-<left>**
		• **<f1> C-S-<down>**
		• **<f1> C-S-<up>**
		
		• **<f11> C-S-<right>**
		• **<f11> C-S-<left>**
		• **<f11> C-S-<down>**
		• **<f11> C-S-<up>**
		
		• **<f7> C-S-<right>**
		• **<f7> C-S-<left>**
		• **<f7> C-S-<down>**
		• **<f7> C-S-<up>**
			• **pel-close-window-right)**
			• **(pel-close-window-left)**
			• **(pel-close-window-down)**
			• **(pel-close-window-up)**
				Kill window pointed by the cursor's direction.
				
				• The 4 different commands and shown in the same cell for convenience, one for each of the available cursors: **<right>**, **<left>**, **<down>** and **<up>**.
				• There are 4 possible sets of bindings:
				  • 3 sets of stand-alone commands:
				  • Commands with **<f11>** prefix, always available.
				  • Commands with **ESC** prefix, available when **pel-windmove-on-esc-cursor** user option is on (set to t).
				  • Commands with **<f1>** prefix, available when **pel-windmove-on-f1-cursor** user option is on (set to t).
				  • The Hydra-based commands, with the Hydra activated with any of the key sequences that use the **<f7>** prefix. Available when **pel-use-hydra** user option is set to t.

Zoom/Un-Zoom Window
 	
See: **Windows**
	C-z
	• zoom/un-zoom
		• With several windows showing in a CRiSP frame, typing C-z hides all windows except the current one. Typing C-z again restores the windows to how they were.
		• Emacs does not have the same functionality. It is possible to get something similar using one standard command and one from the winner external package.:
		  • **C-x 1** hides all windows except the current one (effectively doing what CRiSP calls a zoom). But typing it again does not restore it.
		  • To restore the windows the way they were before you need to use winner-undo from the **winner** built-in package. This is bound to **<f11> w p**
		  • You can also go back in the other history direction with winner using the winner-redo.		

	Close all other windows	• **C-x 1** • **<f7> 1** • **<f7> .**	(**delete-other-windows** &optional WINDOW)	Make current window fill its frame.
	Restore an earlier window configuration	• **C-c <left>** • **<f11> w p** • **<f7> p**	(**winner-undo**)	Switch back to an earlier window configuration saved by Winner mode. In other words, "undo" changes in window configuration.
	Restore a more recent window configuration	• **C-c <right>** • **<f11> w n** • **<f7> n**	(**winner-redo**)	Restore a more recent window configuration saved by Winner mode.
Searching for text in a buffer See: **Search/ Replace**	**<f5>** • search for a string	• Emacs has a **lot** of string search facilities. CRiSP uses the CRiSP regular expression. I have not found anything that support CRiSP regular expressions. Emacs has its own regular expression syntax and also support PCRE. • The main search mechanism is C-s witch is a literal but incremental search. • M-C-s provides a regular expression incremental search. • The direction of the search can be changed during the search. Otherwise the C-r and C-M-r start the searches backward. • The way the search results are displayed can also be changed. You can get them displayed on the window, or a list with further ability to refine the search with all sorts of criteria if you use ivy or helm mode. • Note that it is possible to perform operations during an incremental reach, such as changing the case sensitivity, the way words are treated, etc… • ⚠️ Also note that newlines are NOT described as \n in Emacs: to specify a newline in a search or replace you must insert a new line in your seared text and you use **C-q C-j** for that. • All the information is in the **Search/Replace** table. I'm just copying the main commands here.		

	ISearch - forward
	
	• **Incremental**
	  • **literal search**
	  • **regexp search**
	• Captures string searched,
	• search again with **C-s** or **C-r**
		• **C-s**
		• **⌘-f**
			(**isearch-forward** &optional REGEXP-P NO-RECURSIVE-EDIT)	Do incremental search forward: start or continue a search.
				📦 On PEL: this key mapping is used when either **pel-initial-search-tool** nil or 'anzu' when **pel-use-anzu** is **t**.
				  • If **pel-use-swiper** is **t**, you can use **<f11> s s** to change the tool used for search operations.

		• With a prefix argument, do an **incremental regular expression search** instead, something like:
		  • **C-u 1 C-s**
		  • **M-- C-s**
		  • With PEL, **C-- C-s** works.
		  • **C-u C-s** does **not** work to perform a regexp ISearch.
		  ☛ Instead you can also use **C-M-s** to perform the regexp incremental search forward.
		• To continue to next match during search: type **C-s** again (with prefix argument if that was used for regexp Isearch).
		• To change direction: type **C-r**
		• To repeat last completed incremental search forward: **C-s C-s**
		• **⌘-f** is always mapped to isearch-forward.
		• When Anzu is used (see below) the modeline shows the match count.		

	ISearch - backward
	
	• **Incremental**
	  • **literal search**
	  • **regexp search**
	• Captures string searched,
	• search again with **C-s** or **C-r**
		C-r	(**isearch-backward** &optional REGEXP-P NO-RECURSIVE-EDIT)	Do incremental search backward: start or continue a search.
				📦 On PEL: this key mapping is used when either **pel-initial-search-tool** nil or 'anzu' when **pel-use-anzu** is **t**.
				  • If **pel-use-swiper** is **t**, you can use **<f11> s s** to change the tool used for search operations.

		• With a prefix argument, do an **incremental regular expression search** instead; something like:
		  • **C-u 1 C-r**
		  • **M-- C-s**
		  • With PEL, **C-- C-r** works.
		  • **C-u C-r** does **not** work to perform a regexp ISearch.
		  ☛ Instead you can also use **C-M-r** to perform the regexp incremental search forward.
		• To continue to next match during search: type **C-r** again (with prefix argument if that was used for regexp Isearch).
		• To change direction: type **C-s**
		• To repeat last previously completed incremental search backward: **C-r C-r**
		• When Anzu is used (see below) the modelling shows the match count.		

	ISearch - Regexp — forward
	• **Incremental**
	  • **regexp search**
		C-M-s	(**isearch-forward-regexp** &optional NOT-REGEXP NO-RECURSIVE-EDIT)	Incremental forward regular expression search.
				  ☛ Everything that can be done with **C-s** can also be done here. For example repeating the search can be done with **C-s**.

Concept	CRiSP key	Emacs Key	command	Description
	ISearch - Regexp - backward • **Incremental** • **regexp search**	**C-M-r**	(**isearch-backward-regexp** &optional NOT-REGEXP NO-RECURSIVE-EDIT)	Incremental backward regular expression search. ☛ Everything that can be done with **C-r** can also be done here. For example repeating the search can be done with **C-r**.
Search Again See: ⨌ Search/ Replace	**S-<f5>** search again	To repeat a search in Emacs type **C-s**		
	ISearch - forward • search again with **C-s** or **C-r**	• **C-s** • ⌘-f		Do incremental search forward: start or **continue** a search forward. Any search, including one done with the command described below (**<f11> s .**)
	ISearch - backward • search again with **C-s** or **C-r**	**C-r**		Do incremental search backward: start or **continue** a search backward. Any search, including one done with the command described below (**<f11> s .**)

Search word from top of window or window below
 	
See:
• ⨌ Search/ Replace
• Emacs Numeric Arguments
	C-y
	• search word from top of window or window below
		• In CRiSP with CRiSPer, C-y takes the word the begins after the cursor and search for this word from the top of the buffer in the current window or, if there is a window below, from the top of the buffer in the window below.
		• In Emacs with PEL, for something similar (but more flexible, see below) you can type one of the 2 key sets:
		  • the **<f11> s .** key sequence
		  • the **.;** key-chord, when the key-chord is activated: you must type the 2 keys **.** and **;** together, at the same time.
		• Emacs is much more flexible because Emacs supports the ability to provide **key numeric arguments** to a command. Read the **Emacs manual section on numeric arguments** to understand the concept.		

	Search for:
	• **text in marked region or,**
	• **word taken at point**
	from the top of current or specified window
	
	See also: ⨌ Key-Chords
	
	Note:
	• Captures string searched,
	• Search again with **C-s** or **C-r**
	• Supports toggling the word mode when grabbing word at point.
		• **<f11> s .**
		• **.;**
			(**pel-search-word-from-top** &optional N)	Search for text in marked region or word at point from top/bottom of buffer of window identified by the number of non-dedicated windows and by the numeric argument N.
				• A numeric argument is composed with the Meta key prior to the command:
				• For example, to search a word in the buffer of window located at the right of the current one, position the point on the word to search and type one of the following key sequences:
				  • **M-6 <f11> s .**
				  • **M-6 .;**
				📦 With PEL, the **.;** key-chord is also available when **pel-use-key-chord** is non-nil.
				☝Command numeric prefix is available with the key-chord binding. See ⨌ Key-Chords

		• **Search direction**:
		• If there is only one window: search from the top of current buffer.
		• If there is 2 non-dedicated windows, the behaviour depends on the value of the **pel-search-from-top-in-other** user option:
		  • if **pel-search-from-top-in-other** user option is nil (the default) : search from the top of current buffer unless a numeric argument is specifying another window (see below).
		  • if the **pel-search-from-top-in-other** user option is t, search from the top of the ***other*** window unless a numeric argument 3 or 5 is specified, in which case it searches from the top of the current buffer.
		• If there are 3 or more non-dedicated windows search into the buffer of the window identified by the numeric argument N (see below).
		  • If N is negative: perform a isearch-backward from the bottom of the buffer in the window selected by the absolute value of N.
		• **Window selection**:
		  • If N is not specified, nil, 1, 3, 7 or 9 and larger: search in current window.
		  • If N is 0: : search in other window
		  • If N in [2,8] range, search in window identified by the direction corresponding to the cursor in a numeric keypad:
		  8 := 'up
		  4 := 'left 5 := 'current 6 := 'right
		  2 := 'down
		• **Temporary word mode toggle**: detecting a '*word*' is affected by the subword-mode and superword-mode. When searching in current buffer, the following values of N temporary toggle the mode when grabbing the word:
		  • If N is in [10..18] range: temporary toggle **subword-mode** to grab the word in current buffer, use the N-10 value to identify the window to search.
		  • If N is in [20..28] range: temporary toggle **superword-mode** to grab the word current buffer, use the N-20 value to identify the window to search.
		  • In several major modes (but not all) using **superword-mode** allows you to grab complete identifiers (function names, variables that use embedded underscore or hyphen in their names).
		  • If you do not want to change the mode but want to search for the word as interpreted by the other state of the mode and search in the current buffer, type the command with N in the 20 to 28 range. To toggle superword-mode and search in window above, use: **M-28 <f11> s .**
		• Explicitly selecting the minibuffer window, or a non-existing window is not allowed, and search is done in current window.
		• Searched word is remembered and can be used again to repeat an interactive search with **C-s** or **C-r**.
		• Position before searched word is pushed on the mark ring.	

Replace Text
 	
See: ⨌ Search/ Replace
	<f6>
	• translate
		• Emacs calls this "replacing" text as oppose to translate. The Emacs term is more appropriate.
		• CRiSP text replacement always uses regular expression. Emacs provides literal replacements as well as regular expressions ones.
		• Emacs regular expression syntax differs from CRiSP. Emacs provides tools to test regular expressions that can be very useful. See the details in the ⨌ Search/Replace table.
		• Emacs support query replace (similar to what CRiSP supports) and unconditional replace (where the replacement is done without prompting).
		The response to each query can be one of the following keys:
		  • `y or SPC` : replace
		  • `n or <DEL>` : don't replace, move to next
		  • `.` : replace current and quit
		  • `,` : replace & let me see result before moving on — Press SPC to move on.
		  • `!` : replace all the rest and don't ask
		  • `^` : back up to the previous instance
		  • `u` : undo last replacement
		  • `U` : undo ALL replacements
		  • `q or <RET>` : abort/exit query-replace
		  • `E` : modify the replacement string
		  • `C-r` : enter recursive edit - Exit the recursive edit with one of: **C-M-c** or **C-]**
		  • `C-w` : delete this instance and enter recursive edit —to make a custom replacement
		  • `C-M-c` : exit recursive edit and resume query-replace
		  • `C-]` : Exit recursive edit and exit query-replace
		  • `?` : get help
		  • `Y` : replace all strings in all buffer, no questions. — Multi-buffer QR Response
		  • `N` : skip to next buffer without replacing remaining matches in current buffer — Multi buffer QR Response.		

	Query Replace	**M-%**	(**query-replace** FROM-STRING TO-STRING &optional DELIMITED START END BACKWARD REGION-NONCONTIGUOUS-P)	Replace *some* occurrences of a string with another, both specified by user. A negative argument replaces backwards. ☝When prompted for replacement use M-p to retrieve the original text that you can then modify.

Concept	CRiSP key	Emacs Key	command	Description
	Query Replace Regexp	• `C-M-%` • `<f11> s x q` • `C-c q`	(**query-replace-regexp** REGEXP TO-STRING &optional DELIMITED START END BACKWARD REGION-NONCONTIGUOUS-P) — (**pel-query-replace-string**)	Replace *some* occurrences of a regex match with a specified string. • A negative argument replaces backwards. • C-M-% does not work in Terminal mode. PEL only activates the `C-c q` binding if pel-bind-keys-for-regexp user option is set to **t**. With PEL, when any of pel-use-visual-regexp or pel-use-visual-regexp-steroids is set to t , you can select a regexp engine provided by these external package (using `<f11> s S` to select another) and it affects what command is used here (pel-query-replace-string uses the command corresponding to your selection).
	Unconditional replace	`<f11> s r`	(**replace-string** FROM-STRING TO-STRING &optional DELIMITED START END BACKWARD)	Replace all instances of from-string by to-string from point to end of buffer. Emacs displays the number of string replaced after the operation.
	Unconditional regex replace	• `<f11> s x r` • `C-c r`	(**replace-regexp** REGEXP TO-STRING &optional DELIMITED START END BACKWARD)	Replace every match for regex with new string. PEL only activates the `C-c r` binding if the pel-bind-keys-for-regexp user option is set to **t**. With PEL, when any of pel-use-visual-regexp or pel-use-visual-regexp-steroids is set to **t** , you can select a regexp engine provided by these external package (using `<f11> s S` to select another) and it affects what command is used here (pel-replace-string uses the command corresponding to your selection). It's possible to use lisp expressions in the replacement string, making this super powerful. See **examples in the Emacs Wiki.**
	Start query replace during an incremental seach	`C-s M-%`	(**isearch-query-replace** &optional ARG REGEXP-FLAG)	Transforms the Search into a query replace, using the current string as the string to be replaced. To **replace char at point**, do: `C-s`, `C-M-y` then `M-%` To **replace word at point**, do: `C-s`, `C-w` then `M-%` To **replace line at point**, do: `C-s`, `C-y` then `M-%` You can repeat the middle command to include several chars, words or lines. When prompted for replacement use M-p to retrieve the original text that you can then modify.
Recording/ Playing Keyboard macros See: ∑ Keyboard Macros	**<f7>** • start/stop macro **<f8>** • play macro	• With Emacs you use <f3> to start macro recording and stop the recording with <f4>. • To execute the recorded keyboard macro you use <f4> again (and as many times as required. • Emacs provides the ability to use other keys, to save, name and record keyboard macros as well. • Also, if you record a macro you can type <f3> to insert a count in the generated text, so when you play the macro back each instance will have a different, incrementing count. • More information is available in the ∑ Keyboard Macros table		
	Start Recording	• `<f3>` • `C-x (`	• (**kmacro-start-macro-or-insert-counter** ARG) • (**pel-kmacro-start-macro-or-insert-counter** ARG)	Record subsequent keyboard input, defining a keyboard macro. The commands are recorded even as they are executed.While already defining a macro (with a previous F3), typing F3 inserts the current value of the keyboard macro counter into the buffer, and increments the counter by 1). See The Keyboard Macro Counter. • **C-u <f3>** executes the last macro then appends the keystrokes to its definition. • **C-u C-u <f3>** appends keys to the last defined macro without executing it. By default , the PEL version of the command prompts if a macro already exists, before allowing overwriting it. • Use a negative argument (**M- -** or **C-_**) argument or numeric 0 to prevent this prompt and allow overwriting already defined macro. • This behaviour is customizable. Customize the ***pel-kbmacro-prompts*** variable in the **Pel/Pel Kbmacro** subgroup to change it and prevent the prompting.
	End Recording or call last macro	• `<f4>` • `C-x e`	(**kmacro-end-or-call-macro** ARG &optional NO-REPEAT)	Ends macro recording done with **<f3>**. Typing **<f4>** again runs the last recorded macro. This is the most convenient way to replay a recently recorded macro. Typing **C-u <f4>** runs the *second* macro in the ring. • A prefix argument number N specified the number of times to execute the macro. ⚠ If N is 0 the macro will run forever until it exits with an error (such as encountering the end of the buffer) or it is manually stopped with **C-g** (or **C-<BREAK>** on DOS/Windows)! During that time the display may not even be updated!!
Execute OS Command See: ∑ Shells	**<f10>** • execute command	Emacs provide several commands, listed below. You can run commands synchronously or asynchronously or even run a command taking information from marked lines in a buffer. Or laucha nan application. See ∑ Shells for more info.		
	Run a shell command	• `M-!` • `⌘-L`	(**shell-command** COMMAND &optional OUTPUT-BUFFER ERROR-BUFFER)	Prompts for the command in the minibuffer, show the command output in the next window in the *Shell Command Output* buffer in Fundamental mode.
	Run a command on a marked region	`M-|`	(**shell-command-on-region** START END COMMAND &optional OUTPUT-BUFFER REPLACE ERROR-BUFFER DISPLAY-ERROR-BUFFER)	Execute string COMMAND in inferior shell with region as input. • Normally display output (if any) in temp buffer '*Shell Command Output*'; • Prefix arg means replace the region with it. Return the exit code of COMMAND. • Mark the region first. Then type **M-|**. Emacs prompts for the command to run. Use an argument to replace the region with the command output (ie. type **C-u M-|**)
	Run a shell command asynchronously	`M-&`	(**async-shell-command** COMMAND &optional OUTPUT-BUFFER ERROR-BUFFER)	Execute string COMMAND asynchronously in background. • Like 'shell-command', but adds '&' at the end of COMMAND to execute it asynchronously. • The output appears in the buffer '*Async Shell Command*'. • That buffer is in shell mode.
	Launch OS application	`<f11> A`	(**counsel-linux-app** &optional ARG)	Launch a Linux desktop application, similar to Alt-<F2>. When ARG is non-nil, ignore NoDisplay property in *.desktop files. On Linux, requires the **counsel** external package. PEL activates it when the **pel-use-counsel** user option is set to **t**.
Top of Window See: ∑ Navigation	**C-t** • place line where cursor is to top of screen,	• With CRiSP **C-t** moves current line to the top of window and **C-b** to the bottom. • Emacs uses only one command for both, bound to `C-l` (Control-ell). Type it once: it enters the line, type if again it moves the line to the bottom, type it again it moves the line to the top of the window.		

Concept	CRiSP key	Emacs Key	command	Description
	Position current line to window's Center / Bottom / Top . **Refresh screen.**	`C-l`	(**recenter-top-bottom** &optional ARG)	Without argument: moves the current line to window: center -> top -> bottom. • With arg: centre first: • `C-u C-l` `C-l` `C-l` `C-l` • ➜ center ➜ bottom ➜ center ➜ top • With negative arg: bottom first: • `C-- C-l` `C-l` `C-l` • ➜ bottom ➜ center ➜ top • With arg 0: top first: • `M-0 C-l` `C-l` `C-l` • ➜ top ➜ bottom ➜ center • With numeric positive: move current line to window top position N • With negative numeric: move current line to bottom window position: -1 := last line
	Reposition comment/ definition in full view	• `C-M-l` • `C-[ C-l` • `Esc C-l`	(**reposition-window** &optional ARG)	Attempts to make the current comment or current definition fully visible by scrolling the lines without changing the point. • Further invocations move it to the top of the window or toggle the visibility of comments that precede it (by scrolling the lines).
Undo See ⟩⟩ **Undo/Redo/ Repeat/Arg**	**Alt-u** • undo	• CRiSP supports an omnipotent undo: it undoes everything including movements in buffer. • Emacs undo is different: it undoes changes to buffers, not movement. The default Emacs undo can also undo an undo, making it redo. This is confusing to many newcomers to Emacs so it's possible to use the undo-tree external package which separates the undo from redo and assigns 2 different commands. PEL provides user option to select which mechanism to use. You can also restrict undo operation to a region of a buffer. • See ⟩⟩ **Undo/Redo/Repeat/Arg** for more information.		
	Undo : pel-use-undo-tree = nil	• `C-/` • `C-x u` • `M-u` • `C-z` • `s-z` • `⌘-z` • `<f11> u u`	(**undo** &optional ARG)	Undo last changes **using standard Emacs** undo. Also used to undo an undo, causing a redo! • A numeric ARG serves as a repeat count. PEL uses it when the **pel-use-undo-tree** user option is nil (the default). If you are not familiar with standard Emacs undo, please first read about it before using it. • It might seems strange at first to use the same key to undo and redo.
	Undo : pel-use-undo-tree = t		(**pel-undo** &optional ARG) • (**undo-tree-undo** &optional ARG) • (**undo** &optional ARG)	Undo changes. Does not redo. • A numeric ARG serves as a repeat count. • In Transient Mark mode when the mark is active, only undo changes within the current region. Similarly, when not in Transient Mark mode, just **C-u** as an argument limits undo to changes within the current region. • `C-/` only works in graphics mode • `s-z` and `⌘-z` only work in macOS graphic mode. Note: with PEL, `⌘-z` is `s-z`. PEL uses this when the **pel-use-undo-tree** user option is t. PEL uses the **undo-tree package** instead of the default undo. With PEL, when **pel-use-undo-tree** is **t**, this key is bound to **pel-undo** which uses **undo-tree-undo** by default. • You can, however toggle the local or global **undo-tree-mode** by issuing the `M-x global-undo-tree-mode` or `M-x undo-tree-mode`. If the undo-tree-mode is not set in the buffer, PEL will use the Emacs standard **undo** command until the undo-tree-mode is re-enabled.
	Redo : pel-use-undo-tree = t	• `M-U` • `<f11> u r` • `s-Z` • `⌘-Z`	(**pel-redo** &optional ARG) • (**undo-tree-redo** &optional ARG) • (**undo** &optional ARG)	Redo changes. A numeric ARG serves as a repeat count. • In Transient Mark mode when the mark is active, only redo changes within the current region. Similarly, when not in Transient Mark mode, just **C-u** as an argument limits redo to changes within the current region. • `s-Z` and `⌘-Z` only works in graphics mode • Note: with PEL, `⌘-Z` is `s-Z`. PEL uses the **undo-tree package** instead of the default undo. Under PEL activate the undo-tree package by setting the **pel-use-undo-tree** user option to **t**. With PEL, when **pel-use-undo-tree** is **t**, this key is bound to **pel-redo** which uses **undo-tree-redo** by default. • You can, however toggle the local or global undo-tree-mode by issuing the `M-x global-undo-tree-mode` or `M-x undo-tree-mode`. If the undo-tree-mode is not set in the buffer, PEL will use the Emacs standard **undo** command until the undo-tree-mode is re-enabled.
	Show undo tree : pel-use-undo-tree = t	`<f11> u v`	(**undo-tree-visualize**)	Show undo tree of current buffer. The *undo tree* keys are: • `<up>/<down>` : move up/down the undo tree nodes • `<right>/<left>` : changes branch when at a branch root • `s` : toggle selection mode: normally moving restores right away, this other mode allows you to move in the tree without changing the controlled buffer until RET is typed. • `d` : shows diff between buffer and currently selected undo node!! • `t` : toggles showing relative timestamp on undo nodes PEL uses the **undo-tree package** instead of the default undo. Under PEL activate the undo-tree package by setting the **pel-use-undo-tree** user option to **t**. With PEL, this is available when pel-use-undo-tree is **t** but also while the global or local undo-tree-mode is active, which it should be unless you explicitly disabled one of these via the global-undo-tree-mode or undo-tree-mode commands. If that is the case, re-enable the undo-tree-mode and you will be able to use the command.
	Switch branch of undo tree : pel-use-undo-tree = t	`<f11> u x`	(**undo-tree-switch-branch** BRANCH)	Switch to a different BRANCH of the undo tree. • This will affect which branch to descend when *redoing* changes using 'undo-tree-redo'. PEL uses the **undo-tree package** instead of the default undo. Under PEL activate the undo-tree package by setting the **pel-use-undo-tree** user option to **t**. With PEL, this is available when pel-use-undo-tree is **t** but also while the global or local undo-tree-mode is active, which it should be unless you explicitly disabled one of these via the global-undo-tree-mode or undo-tree-mode commands. If that is the case, re-enable the undo-tree-mode and you will be able to use the command.

Concept	CRiSP key	Emacs Key	command	Description
	Goto last change	**<f11> u **	(**goto-last-change** &optional MARK-POINT MINIMAL-LINE-DISTANCE)	Set point to the position of the last change. • Consecutive calls set point to the position of the previous change. • With a prefix arg (optional arg MARK-POINT non-nil), set mark so **C-x C-x** will return point to the current position. 📦 This requires the **goto-last-change.el** package. Under PEL set the **pel-use-goto-last-change** user option to activate this.

Marking
 	
See ⟫ **Marking**
	CRiSP has 4 commands for marking:
	• Alt-a and Alt-m mark text from a character to where the cursor moves to including/excluding the cursor.
	• Alt-l marks the current line
	• Alt-c marks a rectangular region of text.
	
	• CRiSP allows the cursor to move freely over void area: area where there is no character in the buffer.
	
	• By Default Emacs does not allow that: the cursor can move up to the end of the line; moving vertically will be restricted to the length of the new current line. This affects the way marking is done in Emacs.
	• Emacs also manages 2 position in the buffer:
	  • the point: the location of the cursor.
	  • the mark: the location of the other end of a baked area, often called the "*region*" or "*marked region*". This region may exist and NOT be highlighted. It is highlighted when Emacs buffer uses the minor mode called the "**Transient Mark Mode**", enabled by default, which highlights the region when the mark is active.
	• Emacs also maintains past mark positions inside mark-ring stack structures:
	  1. One **local mark ring** per buffer. The mark ring is a list of positional elements called "*markers*".
	  The maximum length of each mark ring is controlled by the "*mark-ring-max*" customizable variable which is 16 by default.
	  2. One **global mark ring**, which holds the markers of the marks set inside each buffer last visited.
	  The maximum length of that global mark ring is controlled by the customizable *"global-mark-ring-max"* variable which is also 16 by default.
	
	To **cancel a current mar**k, type **C-g**
	
	See the ⟫ **Marking** reference table for more information. Some commands are shown here.			

	Show mark ring stats	• **<f11> . ?** • **<f11> ? .**	(**pel-mark-ring-stats**)	Show info about global and buffer local mark and mark rings; their current and maximum size, buffer and positions for each mark ring entry. • Use it to understand the impact of commands on the mark and mark rings.
	Set mark & activate/ deactivate it	• **C-SPC** • **C-@** • **<f11> . s**	(**set-mark-command** ARG)	Set the mark where point is and toggle its activation. • If mark was not active it activates it: moving the cursor further will show the marked area (the region) if transient mode is enabled (the default in Emacs). • If the mark is active, de-activates it. ☛ Issuing the command twice (**C-SPC C-SPC**) sets the mark location and de-activates it.
	Mark region by semantic unit, increase marked region on each invocation. **★ Powerful command ★**	• **M-=** • **<f11> . =**	(**er/expand-region** ARG)	Increase selected region by semantic units. • With prefix argument expands the region that many times. • If prefix argument is negative calls 'er/contract-region'. • If prefix argument is 0 it resets point and mark to their state before calling 'er/expand-region' for the first time. This command is very powerful: the first time it's typed it selects a word, if you type it again it will expand the selection, and again, and again. The expansions follow the semantics of the current major mode: it is aware of the semantics of several programming languages. ☛ Once **M-=** is typed, you can quickly type the following single keys in sequence: • **=** to expand the region, • **-** to contract the region, • **0** to reset the operation. If you wait too long, then you have to use **M-=** again to continue the expansion, otherwise the region is de-activated. Note that you can also use the following key chords to control the contraction of the selected text without having to worry about time: • **M- M-=** to contract the region • **M-0 M-=** to reset the operation. • Also you can use the cursor keys to expand or contract the region and **C-x C-x** to exchange mark and point to expand the other side of the region with cursors. ✂ **M-=** is normally assigned to count-words-region. PEL binds **<f11> c W** to count-words-region instead. 📦 This requires the expand-region package. Under PEL, activated with ***pel-use-expand-region*** user option. ☛ The PEL package uses this command and key binding for it, a popular binding for this command is **C-=** but that key does not work in text terminal mode. The standard Emacs binding for **M-=** is normally **count-words-region** used for counting words in region, but PEL provides **<f11> c r** for that.

Marking text area with navigation key
 	
See ⟫ **Navigation**
	Alt-a/Alt-m
	• mark beginning to including/ excluding cursor for copy/paste
		• CRiSP marks area of text using the Alt-a and Alt-m and cursor movements.
		• Emacs achieve similar functionality by using "Shift marking": Press the Shift-key and use one of the many cursor movements that support shift marking:
		  • All cursor keys without modifiers: <up>, <down>, <left> and <right>
		  • Almost all other navigation commands except for some in terminal mode
		  • Some of the commands are shown below. Much more is listed in the ⟫ Navigation reference table.		

	Previous line	• **C-p** • **<up>**	(**previous-line** &optional ARG TRY-VSCROLL)	Move cursor vertically up ARG lines. • **C-p** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **<up>** : ☛ Shift marking works with this command.
	Next line	• **C-n** • **<down>**	(**next-line** &optional ARG TRY-VSCROLL)	Move cursor vertically down ARG lines. • **C-n** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **<down>** : ☛ Shift marking works with this command.
	left/previous char	**<left>**	(**left-char** &optional N)	Move point N characters to the left (to the right if N is negative). On reaching beginning or end of buffer, stop and signal error. ☛ Shift marking works with this command.
	left/previous char	**C-b**	(**backward-char** &optional N)	Move point N characters backward (forward if N is negative). • On attempt to pass beginning or end of buffer, stop and signal error. • Interactively, N is the numeric prefix argument. • If N is omitted or nil, move point 1 character backward. • Depending on the bidirectional context, the movement may be to the right or to the left on the screen. This is in contrast with <left>. ☛ Shift marking is available in graphics mode, not in terminal mode.
	right/next char	**<right>**	(**right-char** &optional N)	Move point N characters to the right (to the left if N is negative). On reaching beginning or end of buffer, stop and signal error. ☛ Shift marking works with this command.

Concept	CRiSP key	Emacs Key	command	Description
	right/next char	**C-f**	(**forward-char** &optional N)	Move point N characters forward (backward if N is negative). • On reaching end or beginning of buffer, stop and signal error. • Interactively, N is the numeric prefix argument. • If N is omitted or nil, move point 1 character forward. • Depending on the bidirectional context, the movement may be to the right or to the left on the screen. This is in contrast with <right>. ☛ Shift marking is available in graphics mode, not in terminal mode.

Marking rectangle area
 	
See ∑ **Rectangles**
	Alt-c
	• mark column for copy/paste
		• Since CRiSP allows the cursor to move anywhere, even over "void" space (where the position does not correspond to a character inside the buffer), it also marking any are with a horizontal/vertical line or a rectangle by starting the marking with Alt-c and moving the cursor to the opposite end of the line or rectangle.
		• Emacs does not have a command to mark a rectangle indicating that further operations are related to a rectangle area. Emacs supports copying a rectangle area but it must be done using several commands.
		  • To operate (copy, kill, delete) on a rectangle area the user must:
		  1. first define the rectangle area by setting the mark (for example with **C-SPC** or by using Shift marking).
		  2. then move the cursor (point, in Emacs-speak) to the location of the opposite corner of the rectangle.
		  3. Finally use a rectangle copy, kill or delete command: one of the the first 3 commands below.
		  • To paste a rectangle area into the buffer, use the yank-rectangle command (bound to **C-x r y**)
		
		• Also note that Emacs normally does not allow moving the cursor into the "void" space. It can be done, but a special mode must be activated to do so (more on that below).		

	Save rectangle text See also: ∑ **Cut & Paste**	• **C-x r M-w** • **<f11> = r**	(**copy-rectangle-as-kill** START END)	Copy the region-rectangle and save it as the last killed one.
	Kill text in rectangle See also: • ∑ **Cut & Paste**	• **C-x r k** • **<f11> - r**	(**kill-rectangle** START END &optional FILL)	Delete the region-rectangle and save it as the last killed one. • If the buffer is read-only, Emacs will beep and refrain from deleting the rectangle, but put it in 'killed-rectangle' anyway. This means that ou can use this command to copy text from a read-only buffer. (If the variable 'kill-read-only-ok' is non-nil, then this won't even beep.)
	Delete rectangle text	**C-x r d**	(**delete-rectangle** START END &optional FILL)	Delete (don't save) text in the region-rectangle. • The same range of columns is deleted in each line starting with the line where the region begins and ending with the line where the region ends. • With a prefix (or a FILL) argument, also fill lines where nothing has to be deleted.
	Yank last killed rectangle	**C-x r y**	(**yank-rectangle**)	Yank the last killed rectangle with upper left corner at point.

Marking complete lines
 	
See ∑ **Marking**
	Alt-l
	• mark line
		• CRiSP uses **Alt-l** (ell) to mark a complete line, regardless of the column location of the cursor.
		• Emacs does not explicitly support that concept.
		  • However, PEL does add commands for that, using the concept of "Shift marking", using the Shift key with other keys. They are listed below. With these commands the cursor position does not matter: the entire line is marked. Note that you can also mark a line with S-<down> and <S-up> but the cursor must be at the beginning of the line to mark the entire line.
		  • More information is available in the ∑ **Marking** reference table.		

	Mark line(s) going down	• **M-S-<down>** • **<f11> . <down>**	(**pel-mark-line-down** &optional N)	Mark current line or N line forward for going down. • Set mark at beginning of line, move point to line end. • Without argument select the current line. With numeric argument N, selects the current line and N-1 lines below. ☛ Once the line is marked this way, pressing the same keys or **<down>** key alone grows the region by one more line downward.
	Mark line(s) going up	• **M-S-<up>** • **<f11> . <up>**	(**pel-mark-line-up** &optional N)	Mark current line or N previous lines for going up. • Move point to start of line, set mark at end of line. • Without argument select the current line. With numeric argument N, selects the current line and N-1 lines above. ☛ Once the line is marked this way, pressing the the same keys or **<up>** key alone grows the region by one more line downward.
Listing current buffers See ∑ **Buffers**	**Alt-b** • List file buffers	• CRiSP uses **Alt-b** to pop-up a list of current buffers. Most CRiSP buffers are file buffers, it also supports other buffer types, shells for example. • Emacs handles a large set of buffers: buffer for files, buffers not associated with files (yet), special buffers with no file associated, shells, compilation logs, etc. On a typical Emacs session the number of buffers is normally bigger than the equivalent CRuSP session. Emacs also has buffers that are normally hidden but can be shown if you happen to know their name (which normally have a name that start with a space). • Emacs has several commands to deal with buffers, listing them, change the current buffer. There is also a large number of package that modify and enhance dealing with buffers. Some are shown here, more are listed in the ∑ Buffers reference table.		
	List all buffers	**C-x C-b**	• (**list-buffers** &optional ARG) • (**ibuffer** &optional OTHER-WINDOW-P …)	Display a list of existing buffers in a buffer named "*Buffer List*", the buffer displays information about all buffers and enters the ***Buffer Menu Mode***. See the keystrokes for the Buffer Menu Mode below. ☛ The PEL package the '**ibuffer**' function instead, which provides more functionality, working like dired, allowing to sort by name, size, mode, filtering by mode (hit return on the mode of a buffer). Type **<f1> m** to get the list of possible actions that can be done on the listed buffers.
	Switch to buffer	**C-x b**	(**switch-to-buffer** BUFFER-OR-NAME &optional NORECORD FORCE-SAME-WINDOW)	Switch window to display the previous, or another buffer (entered at prompt). ☝ The invisible buffers have a name that start with a space. To see them type space and tab and a list of those buffers will appear before the list of visible buffers.
	Open Buffer Menu	**<C-f10>**	(**buffer-menu-open**)	Start key navigation of the buffer menu. • Lists the buffers by major-mode when several buffers of the same major-mode are opened. • This is the keyboard interface to <C-down-mouse-1>
Save File See ∑ **File-mngt**	**Alt-w** • save current buffer to file	• CRiSP uses Alt-w to "write" buffer to a file. • Emacs uses C-x C-s to "save" buffer to a file. If the file is locked by another process Emacs prompts and you must "steal" the file to save the buffer to it. The main commands are shown below. But much more is available. See the complete list in the ∑ File-mngt table. Also notice that you can save a file from the ibuffer buffer list command described above.		
	Save file to disk	• **C-x C-s** • **⌘-s**	(**save-buffer** &optional ARG)	Save current buffer to associated file. By default, it makes the previous version into a backup file if previously requested or if this is the first save. • With **C-u**: marks this version to become a backup when the next save is done • With **C-u C-u**: makes the previous version into a backup file • With **C-u C-u C-u:** marks this version to become a backup when the next save is done, and makes the previous version into a backup file. • With prefix 0: never make the previous version into a backup file. On macOS in graphics mode only: **⌘-s** brings a OS file-save dialog.
	Save all/some files	**C-x s**	(**save-some-buffers** &optional ARG PRED)	Prompt for files that are modified. Options: • y : save • n : don't save • C-r : look at the buffer in question • d : view differences with diff-buffer-with-file

Concept	CRiSP key	Emacs Key	command	Description
	Write buffer to specified file	**C-x C-w**	• (**write-file** FILENAME &optional CONFIRM) • (**ido-write-file**)	Similar to "Save-As": prompt for the filename. • Can also be yanked in the mini buffer, use **M-n** to edit it. 💡 Use that command to rename the file.
Deleting/Killing Text See ∑ Cut & Paste	• In CRiSP there is only one concept of deletion: when text is deleted it's gone. You can't get a copy of what was deleted. You can undo the deletion, but not retrieve what was deleted. • Emacs has two different operations: • **Text deletion**. Once the text is deleted it's gone and not retrievable just like in CRiSP. This is not used much in Emacs though. • **Text kill operations**. This is closer to the CUA text cut operation. But with a twist: Emacs killed text is remembered inside the Emacs kill ring, an internal buffer that remembers the text that was killed. Text killed in consecutive operations is concatenated inside a single kill buffer entry. Text killed in operations separated by at least one other non-kill command is stored in new kill ring entries. • The last kill ring entry can be **yanked** back inside a buffer. The main key for that is **C-y** but PEL binds other keys (see below in the Paste section). • Right after a yank operation, you can change the entry yanked with another entry from the kill ring by using the yank-pop command (bound **M-y**). • Emacs provides operation to kill character, word, symbols, S-expressions (text within parentheses) semantic entities of code, paragraphs, code function, entire buffer. See the ∑ Cut & Paste reference table for more information.		
Delete - line See ∑ Cut & Paste	**Alt-d** • Delete line	• In CRiSP, deleting a line is done with Alt-l. • In Emacs, **C-w** kills a line (remembers the kine in the kill buffer). With PEL, if you want to delete the line type **C-- C-w**. Also, with PEL, if there is a region marked, **C-w** will kill that region (or delete it if you type **C-- C-w**).		
	Kill/Delete marked region/line(s) ★PEL Enhanced Key ★ **Available in PEL non numlock mode** See also: • ∑ Marking • ∑ Numkeypad	• **C-w** • **<f11> - l** • **<kp-subtract>** • ⌘-x	(**pel-kill-or-delete-marked-or-whole-line** &optional N)	Flexible region/whole-line kill/delete. Argument controls behaviour (see next cell below). ☝ In graphics mode this also copies text to the OS clipboard. ☝ With PEL in non-numlock mode, the **<keypad-subtract>** (the keypad **-** key) is bound to this command. On macOS in graphics mode only: PEL rebinds ⌘-x from (kill-region) to this command, making this easy to use key able to perform more. ☝ See the ∑ Marking table to mark (select) a text region to use with this command.
		• N=0 := kill region (active/visible or not) • Sign of N selects operation: • positive := kill (default) • negative := delete • Select text to delete/kill based on presence of region: • if a region is marked: kill/delete region's text, • if no region: kill/delete abs(N) lines, start at point. • If operation is to kill 1 line and the line is empty, then delete line instead of killing it. • **Scenarios:** • With no arg: • with no active/visible region: kill current line, but if line is empty delete it. • with an active/visible region: kill region's text. • With arg **0**: (**M-0 C-w**) : kill region's text, whether region is active/visible or not. • With a non zero arg: • With no region active/visible: • With arg **-** : (**M- - C-w**) or (**C- - C-w**) : delete current line • With arg **- 1** : (**M- - 1 C-w**) or (**C- - 1 C-w**) : delete current line • With arg **4**: (**M - 4 C-w**) : kill 4 lines including current one. • With arg **-3**: (**M- - 3 C-w**) : delete 3 lines including current one. • With a region active/visible: • With any negative mark argument: delete the region's text. • With no argument or any positive argument: kill the region's text. ✂ This replaces the standard Emacs binding to kill-region which always kill text between mark and point, even when the region is not marked. When text is killed it is killed with kill-region, so it retains the filtering and kill ring text appending capabilities.		
	Kill whole line	C-S-⌫	(**kill-whole-line** &optional ARG)	Deletes current line (in graphics mode). Use **C-w** instead, it is more flexible, see above.
Delete to end of line See ∑ Cut & Paste	**Alt-k** • delete to end of line	• In CRiSP Alt-k deletes to the end of line. • Emacs provides a command to *kill* to the end of the line, placing the text inside the kill ring. It is mapped to **C-k**. It does not provide a command to *delete* the text although that would be easy to implement. • PEL provides a command to delete text to the end of the line and that is mapped to **C-K** and **<f11> - E**. The **C-K** binding does not work in terminal mode but works when Emacs runs in graphics mode.		
	Kill to end of line	• M-⌦ • **C-k** • **<f11> - e**	(**kill-line** &optional ARG)	Kills from current position to end of line. If no visible characters on it kill through newline. • With prefix argument ARG, kill that many lines from point. • Negative arguments kill lines backward. • With zero argument, kills the text before point on the current line. • If you want to append the killed line to the last killed text, use **C-M-w** before **C-k**. • If the buffer is read-only, Emacs will beep and refrain from deleting the line, but put the line in the kill ring anyway essentially performing a copy to kill ring. M-⌦ is bound to (insert-parentheses &optional ARG) as in M-(in terminal mode. The M-⌦ binding works properly in graphics mode.
	Delete to end of line	• **C-K** • **<f11> - E**	(**pel-delete-to-eol**)	Delete text from cursor to end of line. • Nothing is copied to the kill ring.
Paste See ∑ Cut & Paste	**C-v** • paste	• CRiSP is CUA compliant and uses **C-v** to paste. CRiSP also support the **<insert>** key. • Emacs was designed before CUA was designed and published. It can support it via the cua-mode (see ∑M CUA) but that mode interferes with other operations. For new Emacs users it's often best to learn to use Emacs without it at first. • Instead use the following keys and use the yank command to paste text in. It is bound to **C-y**. Notice that PEL also maps it to the **<insert>** key. • Also remember that Emacs keeps all killed or copied text inside a kill-ring and the yank command retrieves the top entry of the kill ring. That may not be what you wanted to insert. You can then use yank-pop (bound to **M-y**) to replace what you just yanked in by the next entry in the kill-ring. If enabled the pop-up-kill-ring pops a menu of all kill-ring entries and you just select the one you want.		
	Yank last killed into buffer See also: ∑ Numkeypad	• **C-y** • ⌘-v • **<kp-0>**	(**yank** &optional ARG)	Reinsert ("paste") the last stretch of killed text. • More precisely, reinsert the most recent kill, which is the stretch of killed text most recently killed OR yanked. Put point at the end, and set mark at the beginning without activating it. With just C-u as argument, put point at beginning, and mark at end. With argument N, reinsert the Nth most recent kill. ☛ ⌘-v In graphical mode: supports OS clipboard. ☛ With PEL, **<kp-0>** which is also the location of the **<insert>** key on some keyboard, performs the same yank operation when the keypad numlock is off. See ∑ Numkeypad

Concept	CRiSP key	Emacs Key	command	Description
	Replace last yank with previous kill	**M-y**	(**yank-pop** &optional ARG)	Replace just-yanked stretch of killed text with a different stretch. • This command is allowed only immediately after a 'yank' or a 'yank-pop'. At such a time, the region contains a stretch of reinserted previously-killed text. 'yank-pop' deletes that text and inserts in its place a different stretch of killed text. • With no argument, the previous kill is inserted. With argument N, insert the Nth previous kill. If N is negative, this is a more recent kill. • The sequence of kills wraps around, so that after the oldest one comes the newest one. ☞ Also referred to as: "yank next".
	Pop-up menu with kill ring content, to select entry to insert at point.	**<f11> M-y**	(**popup-kill-ring**)	Pop-up a menu that shows all entries in kill ring, allowing insertion of a specified kill ring entry at point. • While the pop-up menu is available, it's also possible to perform interactive search in kill ring text: only matching entries will now show in the pop-up men 📦 Available only in **graphics mode** when popup-kill-ring package and its pre-requisites pos-tip and popup are installed. PEL activates this when the **pel-use-popup-kill-ring** user option is set to **t**. • Use **<f11> – <f2>** to access its customization group.
Copy See ∑ Cut & Paste	**C-c** • copy	• CRiSP is CUA compliant and supports **C-c** to copy a marked area. CRiSPer also supports the **keypad +** to copy marked area. Also CRiSPer binds **Alt-keypad +** to copy the word at the cursor. • Emacs was designed before CUA was designed and published. It can support it via the cua-mode (see ∑M CUA) but that mode interferes with other operations. For new Emacs users it's often best to learn to use Emacs without it at first. • Instead use the following keys to copy text in the kill buffer from where it can be restore (yanked).		
	Copy region or line at point **★PEL Enhanced Key ★** See also: • **∑ Marking** • **∑⌨ Numkeypad**	• **M-w** • **<f11> = l** • **<f11> +** • **<kp-separator>**	(**pel-copy-marked-or-whole-line**)	Flexible copy to kill ring.: copy visible region if any, otherwise copy current line to kill ring. ☛In terminal (TTY) mode the **keypad +** key is interpreted as <kp-separator> on macOS so this key is available. ☛ Replaces standard binding to kill-ring-save which only copies region ☝ See the **∑ Marking** table to mark (select) a text region to use with this command.
				The copy operation is controlled by the (optional) argument: • If N = 0: copy region (regardless of whether it is visible or not. • If a region is active/visible: copy the region's text. • if no region is active/visible copy N lines: • If no argument, (N=1) copy current line. • If N > 0: copy current line and N-1 following lines. • If I < 0: copy current line and N-1 previous lines. All copied lines are complete. The copied text is saved in the kill-ring. All copy operations are performed by `**kill-ring-save**' (the original binding for that key). ☛In graphics mode: text is also copied to the OS clipboard.
	Copy complete word at point See also: • **∑⌨ Numkeypad** • **∑ Text Modes**	• **<f11> = w** • **<C-kp-add>**	(**pel-copy-word-at-point**)	Copy word at point. Shows the text copied in the echo area. ☝ See table **∑ Text Modes** for information on text modes that affects this. • The **<f11> t m ?** command displays the mode and the **<f11> t m** prefix allows modifications of the mode. • See changing the word mode to include or exclude some characters as word delimiters: • subword-mode . To toggle that mode: **<f11> t m b** • superword-mode . To toggle that mode: **<f11> t m p**
	Copy complete symbol at point See also: **∑⌨ Numkeypad**	• **<f11> = .** • **M-+** • **<M-kp-add>**	(**pel-copy-symbol-at-point**)	Copy symbol at point. Syntax depends on the syntax table for the buffer. • Shows the text copied in the echo area. ☝ The syntax of the symbol depends on the major mode used by the current buffer.
	CRiSPer commands	• CRiSPer Esc- commands are available in all file types. The Esc-m and Esc-f are langiugae sensitive but the Esc-d, Esc-i and Esc-I are not and these last ones insert the C pre-processor statement sin any type of files. • Emacs PEL command is also available from any mode, but the **<f12>** prefix is mapped to that command only in c-mode, on other modes you must use the longer **<f11> SPC c** prefix. The commands to insert module header and function header blocks are language sensitive but it is possible to insert them inside another type of file by using the longer prefix. Note, however that if the other mode has different comment style, the other comment style is used. • For example, if you want to insert a C function definition block inside a reStructuredText file you could use **<f11> SPC c <f12> f**. • A better way would be to temporary change the mode to c-mode (using **M-x c-mode**) and then use **<f12> <f12> f** to enter it and then return to rst-mode.		
Insert File Module Header See: • **∑ Inserting Text** • **𝔓ɛ - C**	**Esc-m** • creates module header, etc.	• CRiSPer **Esc-m** inserts a file module header at the top of the file using the comment mechanism appropriate for the type of file and has the ability to write code templates for several programming languages. The layout of the generated code is mostly hard-coded and does not lend itself to being used by various teams having different requirements. • Emacs PEL has a similar mechanism: • a generic file-type agnostic skeleton template • a language-specialized one implemented for some programming and markup languages. It currently supports C, Emacs Lisp, Erlang, reStructuredText. Support for more languages will be developed. • PEL generated skeleton templates are customizable via a set of user option variables to selectively enable various styles or features. The PEL language-specialized tempo skeleton can also be completely replaced by your own code (but not the generic ones). • Read PEL user's manual section related to the skeleton for the programming language of interest: • C : **Controlling PEL Tempo Skeleton for C** .		
	Insert generic file module header blockInsert generic file module header block — Language agnostic	**<f6> h**	(**pel-generic-file-header**)	Insert a file header block at the top of the file. • Works only for buffer visiting a file. • Supports all programming and markup language files that have a dedicated major mode. It is also available in buffers for major modes explicitly supported by the **<f12> <f12>** key prefix. This way, those modes can use two different commands to insert file header blocks, each having its own different format. • It supports several programming and markup language and uses the comment style identified by the file extension. If the comment style is unknown the command prompts for one. • The layout of the entered text is controlled by user options. It is possible to create a user-specified skeleton this command will used instead of the one provided by PEL.

Concept	CRiSP key	Emacs Key	command	Description
	• PEL creates key bindings to invoke the skeletons in the supported major modes, using the same key prefix sequence for each mode: **<f12> <f12>**, with the same key bindings for equivalent concepts (such as file header block) as much as possible. • Several aspects of the PEL Emacs Lisp Source Code Style is controlled by the user options inside the **pel-c-code-style** group. This group can be edited with **<f12> <f2>** from a C mode buffer and include the following options relevant for a module header: • **pel-c-skel-module-header-block-style** : allows selecting a user-define module-header comment block. • **pel-c-skel-comment-with-2-star** : controls the format of C-style continuation comments. • **pel-c-skel-insert-file-timestamp** : set whether an automatically updated timestamp is inserted in the file header block. • **pel-c-skel-use-separators** : set whether blocks use horizontal separator lines. • **pel-c-skel-doc-markup** : identifies the documentation markup used. Currently 'none' and 'Doxygen' are available but not implemented. • **pel-c-skel-insert-module-sections** : set whether the template inserts documentation sections in the comment block documenting the C file. • **pel-c-skel-module-section-titles** : identifies the title of the module sections inserted when pel-c-skel-insert-module-sections is **t**. • **pel-c-skel-with-license** : set whether file header blocks use open source software license text controlled by lice. • **pel-c-use-uuid-include-guards** : If set, include guards using pre-processor symbols made out of the file base name and automatically generated UUID strings are inserted in C header files. • Once a skeleton was just entered (or later by activating the pel-tempo-mode) you can move to the next or previous point of interest (so called *tempo-marks)* with the standard tempo-mode keys **C-c M-f** and **C-c M-b** or some other keys like **C-c .** and **C-c ,**.			
	Insert a file header comment block	**<f12> <f12> h**	(pel-c-file-header)	Insert a large file header the includes sections controlled by the user options in the **pel-c-code-style** customization group and some aspects of the C style currently active. See **some examples in the PEL manual**.
		• Prompts for file purpose and insert a complete file header block with the file name, its purpose, automatically updated timestamp if required by customization, license text if required by customization. • If the file is a C header, inserts a safe and portable C pre-processor #include guard statement that uses a symbol made out of the file base name and an automatically generated UUID string. This eliminates possibility of include header file clash. It is inserted when activated by customization (default is on). • If the file is a C source code file, it inserts a set of code section blocks when activated by customization, potentially separated by horizontal separator lines. The blocks identify the code location of header file inclusion, local type, local variables, code, etc… • Automatically activates the PEL tempo skeleton mode so you can move to the target points where extra text must be entered to complete the template. • Use **C-g** to cancel at any prompt.		
Insert C function header See: • PEL - C	**Esc-f** • creates function header and asks for function name, • creates function header based on function name where cursor is,	• CRiSPer **Esc-f** prompts for a function name and insert a function header with a fixed format. • Emacs PEL has a similar mechanism but the format is customizable via a set of user potions. A fully customized skeleton can also be created and used instead of PEL's default. The appropriate user options are listed below. • Currently the mechanism does not support extracting the function name from the text at point. That might be implemented in the future.		
	• PEL creates key bindings to invoke the skeletons in the supported major modes, using the same key prefix sequence for each mode: **<f12> <f12>**, with the same key bindings for equivalent concepts (such as file header block) as much as possible. • Several aspects of the PEL Emacs Lisp Source Code Style is controlled by the user options inside the **pel-c-code-style** group. This group can be edited with **<f12> <f2>** from a C mode buffer and include the following options relevant for a function header template: • **pel-c-skel-comment-with-2-star** : controls the format of C-style continuation comments. • **pel-c-skel-use-separators** : set whether blocks use horizontal separator lines. • **pel-c-skel-doc-markup** : identifies the documentation markup used. Currently 'none' and 'Doxygen' are available but not implemented. • **pel-c-skel-insert-function-sections** : set whether C function templates are inserted in the function description comment. • **pel-c-skell-function-section-titles** : identifies the title of the C function templates sections inserted when pel-c-skel-insert-function-sections is **t**. • **pel-c-skel-function-define-style** : select the C function comment block style. Several styles are provided: - no special comment - a basic, free-format style to describe the function above its code. - a Man-page style comment block with the sections identified by pel-c-skell-function-section-titles - a user defined tempo skeleton loaded from a user specified file name. See the **source code example.** • **pel-c-skel-function-name-on-first-column** : identifies whether return type is located on the same line as function name or just above. • Once a skeleton was just entered (or later by activating the pel-tempo-mode) you can move to the next or previous point of interest (so called *tempo-marks)* with the standard tempo-mode keys **C-c M-f** and **C-c M-b** or some other keys like **C-c .** and **C-c ,**.			
	Insert a function definition with comment block	**<f12> <f12> f**	(pel-c-function)	Insert a C function definition code and comment template. See **some examples in the PEL manual**.
		• The command prompts for the function name and its purpose. • You can hit return both prompts to specify no text; in that case a tempo skeleton marker is left at the location where the text must be inserted and point is left at the first one. • If you enter a function name, it must be a valid C function name (as far as the syntax is concerned). However leading and trailing whitespace is accepted and trimmed and dash characters ('-') are automatically replaced by underscores ('_') for convenience. • If an invalid name is specified it is erased and you are prompted again. Use **M-p** to bring the old value back. • Prompts for function and purpose maintain separate histories. Use **M-p** and **M-n** to navigate in the histories at the prompt. You can also use the **<up>** and **<down>** keys. • The style of the code inserted is controlled by the user options inside the pel-c-code-style group and the various C style element controls of the CC-mode. • Use **C-g** to cancel at any prompt.		
Insert C #define statement See: • PEL - C	**Esc-d** • insert C #define statement			
	Insert #define	**<f12> <f12> d**	(pel-c-define)	Insert a C pre-processor **#define** statement. • If there is text between the beginning of the line and point, insert the statement on the next line, otherwise insert it on the current line, even if there is text after point (to allow inserting it before the name of the symbol to define).
Insert C #include statement See: • PEL - C	**Esc-i** • insert C #include <> statement			
	Insert #include <.h>	**<f12> <f12> i**	(pel-c-include-lib)	Insert a C pre-processor **#include <>** statement to include a library file. • If there is text between the beginning of the line and point, insert the statement on the next line, otherwise insert it on the current line. • If there is text after point, insert a new line to place that text on the next line. • The .h extension is written between the angle brackets and point left right before the period. The next tempo mark is placed at the end of the line (so **C-c .** move point there).
	Insert #include "h"	**<f12> <f12> I**	(pel-c-include-local)	Insert a C pre-processor **#include ""** statement to include a local file. • If there is text between the beginning of the line and point, insert the statement on the next line, otherwise insert it on the current line. • If there is text after point, insert a new line to place that text on the next line. • The .h extension is written between the angle brackets and point left right before the period. The next tempo mark is placed at the end of the line (so **C-c .** move point there).

Concept	CRiSP key	Emacs Key	command	Description
Insert commented separator line See: • ⟫ **Inserting Text** • ⟫ **Comments**	**Esc-l** • Insert commented separator line	• CRiSPer **Esc-l** (ell) inserts a commented line on the current line, using the current margin for the line length. • Emacs PEL implements something similar, mapped to <f6> l (ell) as well as <f11> i l (ell). It supports several programming and markup language and uses the comment style identified by the file extension. If the comment style is unknown the command prompts for one.		
	Insert commented line See also: ⟫ **Comments**	• **<f11> i l** • **<f6> l**	(**pel-insert-line** &optional LINELEN)	Insert a (commented) line before/at current line. • If point is at the beginning of the line insert it there. • If point is in the middle of a line, move point at beginning of line before inserting it. • The number of dash characters of the line is specified by LINELEN: • If LINELEN is not specified the buffer's fill-column value is used. • It supports several programming and markup language and uses the comment style identified by the file extension. If the comment style is unknown the command prompts for one. ☛ fill-column is customizable and can be used as a file or directory variable.