This document describes the GNU Readline Library, a utility which aids in the consistency of user interface across discrete programs that need to provide a command line interface.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA

Command Line Editing

This chapter describes the basic features of the GNU command line editing interface.

Introduction to Line Editing

The following paragraphs describe the notation used to represent keystrokes.

The text `C-k` is read as `Control-K' and describes the character produced when the Control key is depressed and the `k` key is struck.

The text `M-k` is read as `Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the `k` key is struck. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `k`. Either process is known as *metafying* the `k` key.

The text `M-C-k` is read as `Meta-Control-k' and describes the character produced by *metafying* `C-k`.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see section Readline Init File, for more info).

Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RETURN`. You do not have to be at the end of the line to press `RETURN`; the entire line is accepted regardless of the location of the cursor within the line.

Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use your erase character to back up and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type `C-b` to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with `C-f`.

When you add text in the middle of a line, you will notice that characters to the right of the cursor are `pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor are `pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

`C-b`
: Move back one character.

`C-f`
: Move forward one character.

`DEL`
: Delete the character to the left of the cursor.

`C-d`
: Delete the character underneath the cursor.

Printing characters
: Insert the character into the line at the cursor.

`C-_`
: Undo the last thing that you did. You can undo all the way back to an empty line.

Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-b`, `C-f`, `C-d`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-a`
: Move to the start of the line.

`C-e`
: Move to the end of the line.

`M-f`
: Move forward a word.

`M-b`
: Move backward a word.

`C-l`
: Clear the screen, reprinting the current line at the top.

Notice how `C-f` moves forward a character, while `M-f` moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* (re-inserting) it back into the line. If the description for a command says that it `kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it all. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

Here is the list of commands for killing text.

`C-k`
: Kill the text from the current cursor position to the end of the line.

`M-d`
: Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`M-DEL`
: Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.

`C-w`
: Kill from the cursor to the previous whitespace. This is different than `M-DEL` because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

`C-y`
: Yank the most recently killed text back into the buffer at the cursor.

`M-y`
: Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `C-y` or `M-y`.

Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type `M-- C-k`.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first `digit' you type is a minus sign (-), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the `C-d` command an argument of 10, you could type `M-1 0 C-d`.

Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings installed by default, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an *init* file in your home directory. The name of this file is taken from the value of the environment variable `INPUTRC`. If that variable is unset, the default is `` `~/.inputrc' ``.

When a program which uses the Readline library starts up, the init file is read, and the key bindings are set.

In addition, the `C-x C-r` command re-reads this init file, thus incorporating any changes that you might have made to it.

Readline Init Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a `#` are comments. Lines beginning with a `$` indicate conditional constructs (see section Conditional Init Constructs). Other lines denote variable settings and key bindings.

Variable Settings
: You can change the state of a few variables in Readline by using the `set` command within the init file. Here is how you would specify that you wish to use `vi` line editing commands:

```
set editing-mode vi
```

Right now, there are only a few variables which can be set; so few, in fact, that we just list them here:

`editing-mode`
: The `editing-mode` variable controls which editing mode you are using. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can be set to either `emacs` or `vi`.

`horizontal-scroll-mode`
: This variable can be set to either `On` or `Off`. Setting it to `On` means that the text of the lines that you edit will scroll horizontally on a single screen line when they are longer than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to `Off`.

`mark-modified-lines`
: This variable, when set to `On`, says to display an asterisk (`*') at the start of history lines which have been modified. This variable is `off` by default.

`bell-style`
: Controls what happens when Readline wants to ring the terminal bell. If set to `none`, Readline never rings the bell. If set to `visible`, Readline uses a visible bell if one is available. If set to `audible` (the default), Readline attempts to ring the terminal's bell.

`comment-begin`
: The string to insert at the beginning of the line when the `vi-comment` command is executed. The default value is `"#"`.

`meta-flag`
: If set to `on`, Readline will enable eight-bit input (it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The default value is `off`.

`convert-meta`
: If set to `on`, Readline will convert characters with the eigth bit set to an ASCII key sequence by stripping the eigth bit and prepending an `ESC` character, converting them to a meta-prefixed key sequence. The default value is `on`.

`output-meta`
: If set to `on`, Readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is `off`.

`completion-query-items`
: The number of possible completions that determines when the user is asked whether he wants to see the list of possibilities. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. The default limit is `100`.

`keymap`
: Sets Readline's idea of the current keymap for key binding commands. Acceptable `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-move`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`show-all-if-ambiguous`
: This alters the default behavior of the completion functions. If set to `on`, words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell. The default value is `off`.

`expand-tilde`
: If set to `on`, tilde expansion is performed when Readline attempts word completion. The default is `off`.

- Key Bindings The syntax for controlling key bindings in the init file is simple. First you have to know the name of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the init file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: *function-name* or *macro*

keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

In the above example, `C-u' is bound to the function `universal-argument', and `C-o' is bound to run the macro expressed on the right hand side (that is, to insert the text `>&output' into the line).

"*keyseq*": *function-name* or *macro*

keyseq differs from *keyname* above in that strings denoting an entire key sequence can be specified, by placing the key sequence in double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the special character names are not recognized.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the above example, `C-u' is bound to the function `universal-argument' (just as it was in the first example), `C-x C-r' is bound to the function `re-read-init-file', and `ESC [ 1 1 ~' is bound to insert the text `Function Key 1'. The following escape sequences are available when specifying key sequences:

`\C-`
: control prefix

`\M-`
: meta prefix

`\e`
: an escape character

`\\`
: backslash

`\"`
: "

`\'`
: '

When entering the text of a macro, single or double quotes should be used to indicate a macro definition. Unquoted text is assumed to be a function name. Backslash will quote any character in the macro text, including " and '. For example, the following binding will make `C-x \' insert a single \ into the line:

```
"\C-x\\": "\\"
```

Conditional Init Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are three parser directives used.

`$if`

The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test extends to the end of the line; no characters are required to isolate it.

`mode`

The `mode=` form of the `$if` directive is used to test whether Readline is in `emacs` or `vi` mode. This may be used in conjunction with the `set keymap' command, for instance, to set bindings in the `emacs-standard` and `emacs-ctlx` keymaps only if Readline is starting out in `emacs` mode.

`term`

The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the `=' is tested

against the full name of the terminal and the portion of the terminal name before the first `-'. This allows *sun* to match both *sun* and *sun-cmd*, for instance.

`application`
The *application* construct is used to include application-specific settings. Each program using the Readline library sets the *application name*, and you can test for it. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

- $endif This command, as you saw in the previous example, terminates an `$if` command.
- $else Commands in this branch of the `$if` directive are executed if the test fails.

Bindable Readline Commands

Commands For Moving

`beginning-of-line (C-a)`
Move to the start of the current line.

`end-of-line (C-e)`
Move to the end of the line.

`forward-char (C-f)`
Move forward a character.

`backward-char (C-b)`
Move back a character.

`forward-word (M-f)`
Move forward to the end of the next word. Words are composed of letters and digits.

`backward-word (M-b)`
Move back to the start of this, or the previous, word. Words are composed of letters and digits.

`clear-screen (C-l)`
Clear the screen and redraw the current line, leaving the current line at the top of the screen.

`redraw-current-line ()`
Refresh the current line. By default, this is unbound.

Commands For Manipulating The History

`accept-line (Newline, Return)`
Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)`
Move `up' through the history list.

`next-history (C-n)`
Move `down' through the history list.

`beginning-of-history (M-<)`
Move to the first line in the history.

`end-of-history (M->)`
: Move to the end of the input history, i.e., the line you are entering.

`reverse-search-history (C-r)`
: Search backward starting at the current line and moving `up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)`
: Search forward starting at the current line and moving `down' through the the history as necessary. This is an incremental search.

`non-incremental-reverse-search-history (M-p)`
: Search backward starting at the current line and moving `up' through the history as necessary using a non-incremental search for a string supplied by the user.

`non-incremental-forward-search-history (M-n)`
: Search forward starting at the current line and moving `down' through the the history as necessary using a non-incremental search for a string supplied by the user.

`history-search-forward ()`
: Search forward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

`history-search-backward ()`
: Search backward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

`yank-nth-arg (M-C-y)`
: Insert the first argument to the previous command (usually the second word on the previous line). With an argument *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command.

`yank-last-arg (M-., M-_)`
: Insert last argument to the previous command (the last word on the previous line). With an argument, behave exactly like `yank-nth-arg`.

Commands For Changing Text

`delete-char (C-d)`
: Delete the character under the cursor. If the cursor is at the beginning of the line, there are no characters in the line, and the last character typed was not C-d, then return EOF.

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them.

`quoted-insert (C-q, C-v)`
: Add the next character that you type to the line verbatim. This is how to insert key sequences like `C-q`, for example.

`tab-insert (M-TAB)`
: Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`
: Insert yourself.

`transpose-chars (C-t)`
: Drag the character before the cursor forward over the character at the cursor, moving the cursor forward as well. If the insertion point is at the end of the line, then this transposes the last two characters of the line. Negative argumentss don't work.

`transpose-words (M-t)`

Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word (M-u)`
Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

`downcase-word (M-l)`
Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

`capitalize-word (M-c)`
Capitalize the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

Killing And Yanking

`kill-line (C-k)`
Kill the text from the current cursor position to the end of the line.

`backward-kill-line (C-x Rubout)`
Kill backward to the beginning of the line.

`unix-line-discard (C-u)`
Kill backward from the cursor to the beginning of the current line. Save the killed text on the kill-ring.

`kill-whole-line ()`
Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word (M-d)`
Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word (M-DEL)`
Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`
Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`
Delete all spaces and tabs around point. By default, this is unbound.

`yank (C-y)`
Yank the top of the kill ring into the buffer at the current cursor position.

`yank-pop (M-y)`
Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`
Add this digit to the argument already accumulating, or start a new argument. M-- starts a negative argument.

`universal-argument ()`
Each time this is executed, the argument count is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four. By default, this is not bound to a key.

Letting Readline Type For You

`complete (TAB)`
: Attempt to do completion on the text before the cursor. This is application-specific. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion, and so on.

`possible-completions (M-?)`
: List the possible completions of the text before the cursor.

`insert-completions ()`
: Insert all completions of the text before point that would have been generated by `possible-completions`. By default, this is not bound to a key.

Keyboard Macros

`start-kbd-macro (C-x ()`
: Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro (C-x ))`
: Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro (C-x e)`
: Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`
: Read in the contents of your init file, and incorporate any bindings or variable assignments found there.

`abort (C-g)`
: Abort the current editing command and ring the terminal's bell (subject to the setting of `bell-style`).

`do-uppercase-version (M-a, M-b, ...)`
: Run the command that is bound to the corresoponding uppercase character.

`prefix-meta (ESC)`
: Make the next character that you type be metafied. This is for people without a meta key. Typing `` `ESC f' `` is equivalent to typing `` `M-f' ``.

`undo (C-_, C-x C-u)`
: Incremental undo, separately remembered for each line.

`revert-line (M-r)`
: Undo all changes made to this line. This is like typing the `undo` command enough times to get back to the beginning.

`tilde-expand (M-~)`
: Perform tilde expansion on the current word.

`dump-functions ()`
: Print all of the functions and their key bindings to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.