GNU Readline Library User Interface

Edition 4.2, for `Readline Library` Version 4.2.
Apr 2001

Brian Fox, Free Software Foundation
Chet Ramey, Case Western Reserve University

This document describes the end user interface of 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
59 Temple Place, Suite 330,
Boston, MA 02111 USA

1 Command Line Editing

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

1.1 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 ⟨k⟩ key is pressed while the Control key is depressed.

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 pressed. The Meta key is labeled ⟨ALT⟩ on many keyboards. On keyboards with two keys labeled ⟨ALT⟩ (usually to either side of the space bar), the ⟨ALT⟩ on the left side is generally set to work as a Meta key. The ⟨ALT⟩ key on the right may also be configured to work as a Meta key or may be configured as some other modifier, such as a Compose key for typing accented characters.

If you do not have a Meta or ⟨ALT⟩ key, or another key working as 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 1.3 [Readline Init File], page 4). If your keyboard lacks a ⟨LFD⟩ key, typing ⟨C-j⟩ will produce the desired character. The ⟨RET⟩ key may be labeled ⟨Return⟩ or ⟨Enter⟩ on some keyboards.

1.2 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 ⟨RET⟩. You do not have to be at the end of the line to press ⟨RET⟩; the entire line is accepted regardless of the location of the cursor within the line.

1.2.1 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 mistype a character, and not notice the 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 bare essentials for editing the text of an input line follows.

`C-b` Move back one character.

`C-f` Move forward one character.

⟨DEL⟩ or ⟨Backspace⟩
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-_` or `C-x C-u`
Undo the last editing command. You can undo all the way back to an empty line.

(Depending on your configuration, the ⟨Backspace⟩ key be set to delete the character to the left of the cursor and the ⟨DEL⟩ key set to delete the character underneath the cursor, like `C-d`, rather than the character to the left of the cursor.)

1.2.2 Readline Movement Commands

The above table describes the most basic 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, where a word is composed of letters and digits.

`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.

1.2.3 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. ('Cut' and 'paste' are more recent jargon for 'kill' and 'yank'.)

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. Word boundaries are the same as those used by *M-f*.

M-⟨DEL⟩ Kill from the cursor the start of the previous word, or, if between words, to the start of the previous word. Word boundaries are the same as those used by *M-b*.

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

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*.

1.2.4 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' typed 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`', which will delete the next ten characters on the input line.

1.2.5 Searching for Commands in the History

Readline provides commands for searching through the command history for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, Readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. To search backward in the history for a particular string, type *C-r*. Typing *C-s* searches forward through the history. The characters present in the value of the `isearch-terminators` variable are used to terminate an incremental search. If that variable has not been assigned a value, the ⟨ESC⟩ and *C-J* characters will terminate an incremental search. *C-g* will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line.

To find other matching entries in the history list, type *C-r* or *C-s* as appropriate. This will search backward or forward in the history for the next entry matching the search string typed so far. Any other key sequence bound to a Readline command will terminate the search and execute that command. For instance, a ⟨RET⟩ will terminate the search and accept the line, thereby executing the command from the history list. A movement command will terminate the search, make the last line found the current line, and begin editing.

Non-incremental searches read the entire search string before starting to search for matching history lines. The search string may be typed by the user or be part of the contents of the current line.

1.3 Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings installed by default, it is possible to use a different set of keybindings. Any user can customize programs that use Readline by putting commands in an *inputrc* file, conventionally in his 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.

1.3.1 Readline Init File 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 1.3.2 [Conditional Init Constructs], page 8). Other lines denote variable settings and key bindings.

Variable Settings

You can modify the run-time behavior of Readline by altering the values of variables in Readline using the `set` command within the init file. The syntax is simple:

`set` *variable value*

Here, for example, is how to change from the default Emacs-like key binding to use `vi` line editing commands:

```
set editing-mode vi
```

Variable names and values, where appropriate, are recognized without regard to case.

A great deal of run-time behavior is changeable with the following variables.

`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 insert-comment command is executed. The default value is "#".

completion-ignore-case
: If set to 'on', Readline performs filename matching and completion in a case-insensitive fashion. The default value 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. This variable must be set to an integer value greater than or equal to 0. The default limit is 100.

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

disable-completion
: If set to 'On', Readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to self-insert. The default is 'off'.

editing-mode
: The editing-mode variable controls which default set of key bindings is used. 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'.

enable-keypad
: When set to 'on', Readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys. The default is 'off'.

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

horizontal-scroll-mode
: This variable can be set to either 'on' or 'off'. Setting it to 'on' means that the text of the lines being edited 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'.

`input-meta`
: If set to '`on`', Readline will enable eight-bit input (it will not clear the eighth bit in the characters it reads), regardless of what the terminal claims it can support. The default value is '`off`'. The name `meta-flag` is a synonym for this variable.

`isearch-terminators`
: The string of characters that should terminate an incremental search without subsequently executing the character as a command (see Section 1.2.5 [Searching], page 3). If this variable has not been given a value, the characters ⟨ESC⟩ and *C-J* will terminate an incremental search.

`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.

`mark-directories`
: If set to '`on`', completed directory names have a slash appended. The default is '`on`'.

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

`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`'.

`print-completions-horizontally`
: If set to '`on`', Readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen. The default is '`off`'.

`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`'.

`visible-stats`
: If set to '`on`', a character denoting a file's type is appended to the filename when listing possible completions. The default is '`off`'.

Key Bindings
: The syntax for controlling key bindings in the init file is simple. First you need to find the name of the command that you want to change. The following

sections contain tables of the command name, the default keybinding, if any, and a short description of what the command does.

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

In addition to command names, readline allows keys to be bound to a string that is inserted when the key is pressed (a *macro*).

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`, *M-DEL* is bound to the function `backward-kill-word`, 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).

A number of symbolic character names are recognized while processing this key binding syntax: *DEL*, *ESC*, *ESCAPE*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *RUBOUT*, *SPACE*, *SPC*, and *TAB*.

"*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 again 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 GNU Emacs style escape sequences are available when specifying key sequences:

`\C-`	control prefix
`\M-`	meta prefix
`\e`	an escape character
`\\`	backslash
`\"`	⟨"⟩, a double quotation mark
`\'`	⟨'⟩, a single quote or apostrophe

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

`\a` alert (bell)

`\b` backspace

`\d` delete

`\f` form feed

`\n` newline

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\`*nnn* the character whose ASCII code is the octal value *nnn* (one to three digits)

`\x`*nnn* the character whose ASCII code is the hexadecimal value *nnn* (one to three digits)

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes described above are expanded. Backslash will quote any other character in the macro text, including '"' and '''. For example, the following binding will make '*C-x* \' insert a single '\' into the line:

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

1.3.2 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 four 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 both 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 a particular value. 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
```

`$endif` This command, as seen in the previous example, terminates an `$if` command.

`$else` Commands in this branch of the `$if` directive are executed if the test fails.

`$include` This directive takes a single filename as an argument and reads commands and bindings from that file. For example, the following directive reads from '`/etc/inputrc`':

```
$include /etc/inputrc
```

1.3.3 Sample Init File

Here is an example of an *inputrc* file. This illustrates key binding, variable assignment, and conditional syntax.

```
# This file controls the behaviour of line input editing for
# programs that use the Gnu Readline library.  Existing programs
# include FTP, Bash, and Gdb.
#
# You can re-read the inputrc file with C-x C-r.
# Lines beginning with '#' are comments.
#
# First, include any systemwide bindings and variable assignments from
# /etc/Inputrc
$include /etc/Inputrc

#
# Set various bindings for emacs mode.

set editing-mode emacs

$if mode=emacs

Meta-Control-h:  backward-kill-word Text after the function name is ignored

#
# Arrow keys in keypad mode
#
#"\M-OD":        backward-char
#"\M-OC":        forward-char
#"\M-OA":        previous-history
#"\M-OB":        next-history
#
# Arrow keys in ANSI mode
#
"\M-[D":        backward-char
"\M-[C":        forward-char
"\M-[A":        previous-history
"\M-[B":        next-history
#
# Arrow keys in 8 bit keypad mode
#
#"\M-\C-OD":       backward-char
#"\M-\C-OC":       forward-char
#"\M-\C-OA":       previous-history
#"\M-\C-OB":       next-history
#
# Arrow keys in 8 bit ANSI mode
#
#"\M-\C-[D":       backward-char
#"\M-\C-[C":       forward-char
#"\M-\C-[A":       previous-history
#"\M-\C-[B":       next-history
```

```
C-q: quoted-insert

$endif

# An old-style binding.  This happens to be the default.
TAB: complete

# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word -- insert open and close double quotes
# and move to just after the open quote
"\C-x\"": "\"\"\C-b"
# insert a backslash (testing backslash escapes in sequences and macros)
"\C-x\\": "\\"
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
# Add a binding to refresh the line, which is unbound
"\C-xr": redraw-current-line
# Edit variable on current line.
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# use a visible bell if one is available
set bell-style visible

# don't strip characters to 7 bits when reading
set input-meta on

# allow iso-latin1 characters to be inserted rather than converted to
# prefix-meta sequences
set convert-meta off

# display characters with the eighth bit set directly rather than
# as meta-prefixed characters
set output-meta on

# if there are more than 150 possible completions for a word, ask the
# user if he wants to see all of them
set completion-query-items 150

# For FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
$endif
```

1.4 Bindable Readline Commands

This section describes Readline commands that may be bound to key sequences. Command names without an accompanying key sequence are unbound by default.

In the following descriptions, *point* refers to the current cursor position, and *mark* refers to a cursor position saved by the `set-mark` command. The text between the point and mark is referred to as the *region*.

1.4.1 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 the current or 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.

1.4.2 Commands For Manipulating The History

`accept-line (Newline or Return)`
: Accept the line regardless of where the cursor is. If this line is non-empty, it may be added to the history list for future recall with `add_history()`. If this line is a modified history line, the history line is restored to its original state.

`previous-history (C-p)`
: Move 'back' through the history list, fetching the previous command.

`next-history (C-n)`
: Move 'forward' through the history list, fetching the next command.

`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 currently being entered.

`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 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 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) at point. 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-. or M-_)`
Insert last argument to the previous command (the last word of the previous history entry). With an argument, behave exactly like `yank-nth-arg`. Successive calls to `yank-last-arg` move back through the history list, inserting the last argument of each line in turn.

1.4.3 Commands For Changing Text

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

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

`forward-backward-delete-char ()`
: Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cursor is deleted. By default, this is not bound to a key.

`quoted-insert (C-q or C-v)`
: Add the next character typed 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 arguments have no effect.

`transpose-words (M-t)`
: Drag the word before point past the word after point, moving point past that word as well.

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

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

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

1.4.4 Killing And Yanking

`kill-line (C-k)`
: Kill the text from point 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.

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

`kill-word (M-d)`
: Kill from point 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 point. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`
: Kill the word behind point, 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.

`kill-region ()`
: Kill the text in the current region. By default, this command is unbound.

`copy-region-as-kill ()`
: Copy the text in the region to the kill buffer, so it can be yanked right away. By default, this command is unbound.

`copy-backward-word ()`
: Copy the word before point to the kill buffer. The word boundaries are the same as `backward-word`. By default, this command is unbound.

`copy-forward-word ()`
: Copy the word following point to the kill buffer. The word boundaries are the same as `forward-word`. By default, this command is unbound.

`yank (C-y)`
: Yank the top of the kill ring into the buffer at point.

`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`.

1.4.5 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 ()`

This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing `universal-argument` again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on. By default, this is not bound to a key.

1.4.6 Letting Readline Type For You

`complete (⟨TAB⟩)`

Attempt to perform completion on the text before point. The actual completion performed is application-specific. The default is filename completion.

`possible-completions (M-?)`

List the possible completions of the text before point.

`insert-completions (M-*)`

Insert all completions of the text before point that would have been generated by `possible-completions`.

`menu-complete ()`

Similar to `complete`, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of `menu-complete` steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung (subject to the setting of `bell-style`) and the original text is restored. An argument of *n* moves *n* positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to ⟨TAB⟩, but is unbound by default.

`delete-char-or-list ()`

Deletes the character under the cursor if not at the beginning or end of the line (like `delete-char`). If at the end of the line, behaves identically to `possible-completions`. This command is unbound by default.

1.4.7 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.

1.4.8 Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`
Read in the contents of the *inputrc* 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, M-x, ...)`
If the metafied character *x* is lowercase, run the command that is bound to the corresponding uppercase character.

`prefix-meta (⟨ESC⟩)`
Metafy the next character typed. This is for keyboards without a meta key. Typing '⟨ESC⟩ `f`' is equivalent to typing *M-f*.

`undo (C-_ or 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 executing the `undo` command enough times to get back to the beginning.

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

`set-mark (C-@)`
Set the mark to the point. If a numeric argument is supplied, the mark is set to that position.

`exchange-point-and-mark (C-x C-x)`
Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

`character-search (C-])`
A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.

`character-search-backward (M-C-])`
A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.

`insert-comment (M-#)`
The value of the `comment-begin` variable is inserted at the beginning of the current line, and the line is accepted as if a newline had been typed.

`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. This command is unbound by default.

`dump-variables ()`
: Print all of the settable variables and their values 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. This command is unbound by default.

`dump-macros ()`
: Print all of the Readline key sequences bound to macros and the strings they output. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

1.5 Readline vi Mode

While the Readline library does not have a full set of `vi` editing functions, it does contain enough to allow simple editing of the line. The Readline `vi` mode behaves as specified in the POSIX 1003.2 standard.

In order to switch interactively between `emacs` and `vi` editing modes, use the command *M-C-j* (bound to emacs-editing-mode when in `vi` mode and to vi-editing-mode in `emacs` mode). The Readline default is `emacs` mode.

When you enter a line in `vi` mode, you are already placed in 'insertion' mode, as if you had typed an '`i`'. Pressing ⟨ESC⟩ switches you into 'command' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with '`k`' and subsequent lines with '`j`', and so forth.

Table of Contents