GNU Readline Library

Edition 8.0, for `Readline Library` Version 8.0.
November 2018

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

This manual describes the GNU Readline Library (version 8.0, 30 November 2018), a library which aids in the consistency of user interface across discrete programs which provide a command line interface.

Table of Contents

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 current 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.

Readline remembers the last incremental search string. If two *C-r*s are typed without any intervening characters defining a new search string, any remembered search string is used.

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`. If that file does not exist or cannot be read, the ultimate default is `/etc/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 12). 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. Unrecognized variable names are ignored.

Boolean variables (those that can be set to on or off) are set to on if the value is null or empty, *on* (case-insensitive), or 1. Any other value results in the variable being set to off.

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.

`bind-tty-special-chars`
If set to '`on`' (the default), Readline attempts to bind the control characters treated specially by the kernel's terminal driver to their Readline equivalents.

`blink-matching-paren`
If set to '`on`', Readline attempts to briefly move the cursor to an opening parenthesis when a closing parenthesis is inserted. The default is '`off`'.

`colored-completion-prefix`
If set to '`on`', when listing completions, Readline displays the common prefix of the set of possible completions using a different color. The color definitions are taken from the value of the `LS_COLORS` environment variable. The default is '`off`'.

`colored-stats`
If set to '`on`', Readline displays possible completions using different colors to indicate their file type. The color definitions are taken from the value of the `LS_COLORS` environment variable. The default is '`off`'.

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

`completion-display-width`
The number of screen columns used to display possible matches when performing completion. The value is ignored if it is less than 0 or greater than the terminal screen width. A value of 0 will cause matches to be displayed one per line. The default value is -1.

`completion-ignore-case`
If set to '`on`', Readline performs filename matching and completion in a case-insensitive fashion. The default value is '`off`'.

`completion-map-case`
If set to '`on`', and *completion-ignore-case* is enabled, Readline treats hyphens ('`-`') and underscores ('`_`') as equivalent when performing case-insensitive filename matching and completion. The default value is '`off`'.

`completion-prefix-display-length`
The length in characters of the common prefix of a list of possible completions that is displayed without modification. When set to a

value greater than zero, common prefixes longer than this value are replaced with an ellipsis when displaying possible completions.

`completion-query-items`
: The number of possible completions that determines when the user is asked whether the list of possibilities should be displayed. 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. A negative value means Readline should never ask. 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`', but will be set to '`off`' if the locale is one that contains eight-bit characters.

`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`'.

`echo-control-characters`
: When set to '`on`', on operating systems that indicate they support it, readline echoes a character corresponding to a signal generated from the keyboard. The default is '`on`'.

`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`'.

`emacs-mode-string`
: If the *show-mode-in-prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when emacs editing mode is active. The value is expanded like a key binding, so the standard set of meta- and control prefixes and backslash escape sequences is available. Use the '`\1`' and '`\2`' escapes to begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string. The default is '`@`'.

`enable-bracketed-paste`
: When set to '`On`', Readline will configure the terminal in a way that will enable it to insert each paste into the editing buffer as a single string of characters, instead of treating each character as if it had been read from the keyboard. This can prevent pasted characters from being interpreted as editing commands. The default is '`off`'.

`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`'.

`enable-meta-key`
When set to '`on`', Readline will try to enable any meta modifier key the terminal claims to support when it is called. On many terminals, the meta key is used to send eight-bit characters. The default is '`on`'.

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

`history-preserve-point`
If set to '`on`', the history code attempts to place the point (the current cursor position) at the same location on each history line retrieved with `previous-history` or `next-history`. The default is '`off`'.

`history-size`
Set the maximum number of history entries saved in the history list. If set to zero, any existing history entries are deleted and no new entries are saved. If set to a value less than zero, the number of history entries is not limited. By default, the number of history entries is not limited. If an attempt is made to set *history-size* to a non-numeric value, the maximum number of history entries will be set to 500.

`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`', but Readline will set it to '`on`' if the locale contains eight-bit characters. 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. Built-in `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-move`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command` (`vi-move` is also a synonym); `emacs` is equivalent to `emacs-standard`. Applications may add additional names. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`keyseq-timeout`
Specifies the duration Readline will wait for a character when reading an ambiguous key sequence (one that can form a complete key sequence using the input read so far, or can take additional input to complete a longer key sequence). If no input is received within the timeout, Readline will use the shorter but complete key sequence. Readline uses this value to determine whether or not input is available on the current input source (`rl_instream` by default). The value is specified in milliseconds, so a value of 1000 means that Readline will wait one second for additional input. If this variable is set to a value less than or equal to zero, or to a non-numeric value, Readline will wait until another key is pressed to decide which key sequence to complete. The default value is `500`.

`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.

`mark-symlinked-directories`
If set to '`on`', completed names which are symbolic links to directories have a slash appended (subject to the value of `mark-directories`). The default is '`off`'.

`match-hidden-files`
This variable, when set to '`on`', causes Readline to match files whose names begin with a '`.`' (hidden files) when performing filename completion. If set to '`off`', the leading '`.`' must be supplied by the user in the filename to be completed. This variable is '`on`' by default.

`menu-complete-display-prefix`
If set to '`on`', menu completion displays the common prefix of the list of possible completions (which may be empty) before cycling through the list. The default is '`off`'.

`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`', but Readline will set it to '`on`' if the locale contains eight-bit characters.

`page-completions`
: If set to '`on`', Readline uses an internal `more`-like pager to display a screenful of possible completions at a time. This variable is '`on`' by default.

`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`'.

`revert-all-at-newline`
: If set to '`on`', Readline will undo all changes to history lines before returning when `accept-line` is executed. By default, history lines may be modified and retain individual undo lists across calls to `readline`. 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`'.

`show-all-if-unmodified`
: This alters the default behavior of the completion functions in a fashion similar to *show-all-if-ambiguous*. If set to '`on`', words which have more than one possible completion without any possible partial completion (the possible completions don't share a common prefix) cause the matches to be listed immediately instead of ringing the bell. The default value is '`off`'.

`show-mode-in-prompt`
: If set to '`on`', add a string to the beginning of the prompt indicating the editing mode: emacs, vi command, or vi insertion. The mode strings are user-settable (e.g., *emacs-mode-string*). The default value is '`off`'.

`skip-completed-text`
: If set to '`on`', this alters the default completion behavior when inserting a single match into the line. It's only active when performing completion in the middle of a word. If enabled, readline does not insert characters from the completion that match characters after point in the word being completed, so portions of the word following the cursor are not duplicated. For instance, if this is enabled, attempting completion when the cursor is after the '`e`' in '`Makefile`' will result in '`Makefile`' rather than '`Makefilefile`', assuming there is a single possible completion. The default value is '`off`'.

`vi-cmd-mode-string`
If the *show-mode-in-prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when vi editing mode is active and in command mode. The value is expanded like a key binding, so the standard set of meta- and control prefixes and backslash escape sequences is available. Use the '`\1`' and '`\2`' escapes to begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string. The default is '`(cmd)`'.

`vi-ins-mode-string`
If the *show-mode-in-prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when vi editing mode is active and in insertion mode. The value is expanded like a key binding, so the standard set of meta- and control prefixes and backslash escape sequences is available. Use the '`\1`' and '`\2`' escapes to begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string. The default is '`(ins)`'.

`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. There can be no space between the key name and the colon – that will be interpreted as part of the key name. 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 example above, *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 eight-bit character whose value is the octal value *nnn* (one to three digits)

`\xHH` the eight-bit character whose value is the hexadecimal value *HH* (one or two hex 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, after any comparison operator, extends to the end of the line; unless otherwise noted, 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.

`version` The `version` test may be used to perform comparisons against specific Readline versions. The `version` expands to the current Readline version. The set of comparison operators includes '=' (and '=='), '!=', '<=', '>=', '<', and '>'. The version number supplied on the right side of the operator consists of a major version number, an optional decimal point, and an optional minor version (e.g., '7.1'). If the minor version is omitted, it is assumed to be '0'. The operator may be separated from the string `version` and from the version number argument by whitespace. The following example sets a variable if the Readline version being used is 7.0 or newer:

```
$if version >= 7.0
set show-mode-in-prompt on
$endif
```

`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
```

`variable` The *variable* construct provides simple equality tests for Readline variables and values. The permitted comparison operators are '`=`', '`==`', and '`!=`'. The variable name must be separated from the comparison operator by whitespace; the operator may be separated from the value on the right hand side by whitespace. Both string and boolean variables may be tested. Boolean variables must be tested against the values *on* and *off*. The following example is equivalent to the `mode=emacs` test described above:

```
$if editing-mode == emacs
set show-mode-in-prompt on
$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 system-wide 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.

`previous-screen-line ()`
: Attempt to move point to the same physical screen column on the previous physical screen line. This will not have the desired effect if the current Readline line does not take up more than one physical line or if point is not greater than the length of the prompt plus the screen width.

`next-screen-line ()`
: Attempt to move point to the same physical screen column on the next physical screen line. This will not have the desired effect if the current Readline line does not take up more than one physical line or if the length of the current Readline line is not greater than the length of the prompt plus the screen width.

`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 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. The search string may match anywhere in a history line.

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

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

`history-substring-search-forward ()`
: Search forward through the history for the string of characters between the start of the current line and the point. The search string may match anywhere

in a history line. This is a non-incremental search. By default, this command is unbound.

`history-substring-search-backward ()`
: Search backward through the history for the string of characters between the start of the current line and the point. The search string may match anywhere in a history line. 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. Once the argument *n* is computed, the argument is extracted as if the '`!n`' history expansion had been specified.

`yank-last-arg (M-. or M-_)`
: Insert last argument to the previous command (the last word of the previous history entry). With a numeric argument, behave exactly like `yank-nth-arg`. Successive calls to `yank-last-arg` move back through the history list, inserting the last word (or the word specified by the argument to the first call) of each line in turn. Any numeric argument supplied to these successive calls determines the direction to move through the history. A negative argument switches the direction through the history (back or forward). The history expansion facilities are used to extract the last argument, as if the '`!$`' history expansion had been specified.

1.4.3 Commands For Changing Text

`end-of-file` `(usually C-d)`
: The character indicating end-of-file as set, for example, by `stty`. If this character is read when there are no characters on the line, and point is at the beginning of the line, Readline interprets it as the end of input and returns EOF.

`delete-char (C-d)`
: Delete the character at point. If this function is bound to the same character as the tty EOF character, as *`C-d`* commonly is, see above for the effects.

`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.

`bracketed-paste-begin ()`
: This function is intended to be bound to the "bracketed paste" escape sequence sent by some terminals, and such a binding is assigned by default. It allows Readline to insert the pasted text as a single unit without treating each character as if it had been read from the keyboard. The characters are inserted as if each one was bound to `self-insert` instead of executing any editing commands.

`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. If the insertion point is at the end of the line, this transposes the last two words on the line.

`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.

`overwrite-mode ()`
: Toggle overwrite mode. With an explicit positive numeric argument, switches to overwrite mode. With an explicit non-positive numeric argument, switches to insert mode. This command affects only `emacs` mode; `vi` mode does overwrite differently. Each call to `readline()` starts in insert mode.

  In overwrite mode, characters bound to `self-insert` replace the text at point rather than pushing the text to the right. Characters bound to `backward-delete-char` replace the character before point with a space.

  By default, this command is unbound.

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 from the cursor to the beginning of the current 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.

`unix-filename-rubout ()`
: Kill the word behind point, using white space and the slash character as the word boundaries. 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 nor 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. When displaying completions, Readline sets the number of columns used for display to the value of `completion-display-width`, the value of the environment variable `COLUMNS`, or the screen width, in that order.

`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.

`menu-complete-backward ()`
Identical to `menu-complete`, but moves backward through the list of possible completions, as if `menu-complete` had been given a negative argument.

`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.

`print-last-kbd-macro ()`
: Print the last keboard macro defined in a format suitable for the *inputrc* file.

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-lowercase-version (M-A, M-B, M-x, ...)`
: If the metafied character *x* is upper case, run the command that is bound to the corresponding metafied lower case character. The behavior is undefined if *x* is already lower case.

`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.

`skip-csi-sequence ()`
: Read enough characters to consume a multi-key sequence such as those defined for keys like Home and End. Such sequences begin with a Control Sequence Indicator (CSI), usually ESC-[. If this sequence is bound to `"\e["`, keys producing such sequences will have no effect unless explicitly bound to a readline command, instead of inserting stray characters into the editing buffer. This is unbound by default, but usually bound to ESC-[.

`insert-comment (M-#)`
: Without a numeric argument, the value of the `comment-begin` variable is inserted at the beginning of the current line. If a numeric argument is supplied, this command acts as a toggle: if the characters at the beginning of the line do not match the value of `comment-begin`, the value is inserted, otherwise the characters in `comment-begin` are deleted from the beginning of the line. In either case, 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.

`emacs-editing-mode (C-e)`
: When in `vi` command mode, this causes a switch to `emacs` editing mode.

`vi-editing-mode (M-C-j)`
: When in `emacs` editing mode, this causes a switch to `vi` editing mode.

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 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.

2 Programming with GNU Readline

This chapter describes the interface between the GNU Readline Library and other programs. If you are a programmer, and you wish to include the features found in GNU Readline such as completion, line editing, and interactive history manipulation in your own programs, this section is for you.

2.1 Basic Behavior

Many programs provide a command line interface, such as `mail`, `ftp`, and `sh`. For such programs, the default behaviour of Readline is sufficient. This section describes how to use Readline in the simplest way possible, perhaps to replace calls in your code to `gets()` or `fgets()`.

The function `readline()` prints a prompt *prompt* and then reads and returns a single line of text from the user. If *prompt* is `NULL` or the empty string, no prompt is displayed. The line `readline` returns is allocated with `malloc()`; the caller should `free()` the line when it has finished with it. The declaration for `readline` in ANSI C is

```
char *readline (const char *prompt);
```

So, one might say

```
char *line = readline ("Enter a line: ");
```

in order to read a line of text from the user. The line returned has the final newline removed, so only the text remains.

If `readline` encounters an `EOF` while reading the line, and the line is empty at that point, then `(char *)NULL` is returned. Otherwise, the line is ended just as if a newline had been typed.

Readline performs some expansion on the *prompt* before it is displayed on the screen. See the description of `rl_expand_prompt` (see Section 2.4.6 [Redisplay], page 37) for additional details, especially if *prompt* will contain characters that do not consume physical screen space when displayed.

If you want the user to be able to get at the line later, (with `C-p` for example), you must call `add_history()` to save the line away in a *history* list of such lines.

```
add_history (line);
```

For full details on the GNU History Library, see the associated manual.

It is preferable to avoid saving empty lines on the history list, since users rarely have a burning need to reuse a blank line. Here is a function which usefully replaces the standard `gets()` library function, and has the advantage of no static buffer to overflow:

```
/* A static variable for holding the line. */
static char *line_read = (char *)NULL;

/* Read a string, and return a pointer to it.
   Returns NULL on EOF. */
char *
rl_gets ()
{
```

```
    /* If the buffer has already been allocated,
       return the memory to the free pool. */
    if (line_read)
      {
        free (line_read);
        line_read = (char *)NULL;
      }

    /* Get a line from the user. */
    line_read = readline ("");

    /* If the line has any text in it,
       save it on the history. */
    if (line_read && *line_read)
      add_history (line_read);

    return (line_read);
  }
```

This function gives the user the default behaviour of `TAB` completion: completion on file names. If you do not want Readline to complete on filenames, you can change the binding of the `TAB` key with `rl_bind_key()`.

```
int rl_bind_key (int key, rl_command_func_t *function);
```

`rl_bind_key()` takes two arguments: *key* is the character that you want to bind, and *function* is the address of the function to call when *key* is pressed. Binding `TAB` to `rl_insert()` makes `TAB` insert itself. `rl_bind_key()` returns non-zero if *key* is not a valid ASCII character code (between 0 and 255).

Thus, to disable the default `TAB` behavior, the following suffices:

```
rl_bind_key ('\t', rl_insert);
```

This code should be executed once at the start of your program; you might write a function called `initialize_readline()` which performs this and other desired initializations, such as installing custom completers (see Section 2.6 [Custom Completers], page 50).

2.2 Custom Functions

Readline provides many functions for manipulating the text of the line, but it isn't possible to anticipate the needs of all programs. This section describes the various functions and variables defined within the Readline library which allow a user program to add customized functionality to Readline.

Before declaring any functions that customize Readline's behavior, or using any functionality Readline provides in other code, an application writer should include the file `<readline/readline.h>` in any file that uses Readline's features. Since some of the definitions in `readline.h` use the `stdio` library, the file `<stdio.h>` should be included before `readline.h`.

`readline.h` defines a C preprocessor variable that should be treated as an integer, `RL_READLINE_VERSION`, which may be used to conditionally compile application code depending on the installed Readline version. The value is a hexadecimal encoding of the major and minor version numbers of the library, of the form 0x*MMmm*. *MM* is the two-digit major version number; *mm* is the two-digit minor version number. For Readline 4.2, for example, the value of `RL_READLINE_VERSION` would be `0x0402`.

2.2.1 Readline Typedefs

For readability, we declare a number of new object types, all pointers to functions.

The reason for declaring these new types is to make it easier to write code describing pointers to C functions with appropriately prototyped arguments and return values.

For instance, say we want to declare a variable *func* as a pointer to a function which takes two `int` arguments and returns an `int` (this is the type of all of the Readline bindable functions). Instead of the classic C declaration

```
int (*func)();
```

or the ANSI-C style declaration

```
int (*func)(int, int);
```

we may write

```
rl_command_func_t *func;
```

The full list of function pointer types available is

```
typedef int rl_command_func_t (int, int);
typedef char *rl_compentry_func_t (const char *, int);
typedef char **rl_completion_func_t (const char *, int, int);
typedef char *rl_quote_func_t (char *, int, char *);
typedef char *rl_dequote_func_t (char *, int);
typedef int rl_compignore_func_t (char **);
typedef void rl_compdisp_func_t (char **, int, int);
typedef int rl_hook_func_t (void);
typedef int rl_getc_func_t (FILE *);
typedef int rl_linebuf_func_t (char *, int);
typedef int rl_intfunc_t (int);
#define rl_ivoidfunc_t rl_hook_func_t
typedef int rl_icpfunc_t (char *);
typedef int rl_icppfunc_t (char **);
typedef void rl_voidfunc_t (void);
typedef void rl_vintfunc_t (int);
typedef void rl_vcpfunc_t (char *);
typedef void rl_vcppfunc_t (char **);
```

2.2.2 Writing a New Function

In order to write new functions for Readline, you need to know the calling conventions for keyboard-invoked functions, and the names of the variables that describe the current state of the line read so far.

The calling sequence for a command `foo` looks like

```
int foo (int count, int key)
```

where *count* is the numeric argument (or 1 if defaulted) and *key* is the key that invoked this function.

It is completely up to the function as to what should be done with the numeric argument. Some functions use it as a repeat count, some as a flag, and others to choose alternate behavior (refreshing the current line as opposed to refreshing the screen, for example). Some choose to ignore it. In general, if a function uses the numeric argument as a repeat count, it should be able to do something useful with both negative and positive arguments. At the very least, it should be aware that it can be passed a negative argument.

A command function should return 0 if its action completes successfully, and a value greater than zero if some error occurs. This is the convention obeyed by all of the builtin Readline bindable command functions.

2.3 Readline Variables

These variables are available to function writers.

`char * rl_line_buffer` [Variable]

This is the line gathered so far. You are welcome to modify the contents of the line, but see Section 2.4.5 [Allowing Undoing], page 36. The function `rl_extend_line_buffer` is available to increase the memory allocated to `rl_line_buffer`.

`int rl_point` [Variable]

The offset of the current cursor position in `rl_line_buffer` (the *point*).

`int rl_end` [Variable]

The number of characters present in `rl_line_buffer`. When `rl_point` is at the end of the line, `rl_point` and `rl_end` are equal.

`int rl_mark` [Variable]

The *mark* (saved position) in the current line. If set, the mark and point define a *region*.

`int rl_done` [Variable]

Setting this to a non-zero value causes Readline to return the current line immediately.

`int rl_num_chars_to_read` [Variable]

Setting this to a positive value before calling `readline()` causes Readline to return after accepting that many characters, rather than reading up to a character bound to `accept-line`.

`int rl_pending_input` [Variable]

Setting this to a value makes it the next keystroke read. This is a way to stuff a single character into the input stream.

`int rl_dispatching` [Variable]

Set to a non-zero value if a function is being called from a key binding; zero otherwise. Application functions can test this to discover whether they were called directly or by Readline's dispatching mechanism.

`int rl_erase_empty_line` [Variable]
Setting this to a non-zero value causes Readline to completely erase the current line, including any prompt, any time a newline is typed as the only character on an otherwise-empty line. The cursor is moved to the beginning of the newly-blank line.

`char * rl_prompt` [Variable]
The prompt Readline uses. This is set from the argument to `readline()`, and should not be assigned to directly. The `rl_set_prompt()` function (see Section 2.4.6 [Redisplay], page 37) may be used to modify the prompt string after calling `readline()`.

`char * rl_display_prompt` [Variable]
The string displayed as the prompt. This is usually identical to *rl_prompt*, but may be changed temporarily by functions that use the prompt string as a message area, such as incremental search.

`int rl_already_prompted` [Variable]
If an application wishes to display the prompt itself, rather than have Readline do it the first time `readline()` is called, it should set this variable to a non-zero value after displaying the prompt. The prompt must also be passed as the argument to `readline()` so the redisplay functions can update the display properly. The calling application is responsible for managing the value; Readline never sets it.

`const char * rl_library_version` [Variable]
The version number of this revision of the library.

`int rl_readline_version` [Variable]
An integer encoding the current version of the library. The encoding is of the form 0x*MMmm*, where *MM* is the two-digit major version number, and *mm* is the two-digit minor version number. For example, for Readline-4.2, `rl_readline_version` would have the value 0x0402.

`int rl_gnu_readline_p` [Variable]
Always set to 1, denoting that this is GNU readline rather than some emulation.

`const char * rl_terminal_name` [Variable]
The terminal type, used for initialization. If not set by the application, Readline sets this to the value of the `TERM` environment variable the first time it is called.

`const char * rl_readline_name` [Variable]
This variable is set to a unique name by each application using Readline. The value allows conditional parsing of the inputrc file (see Section 1.3.2 [Conditional Init Constructs], page 12).

`FILE * rl_instream` [Variable]
The stdio stream from which Readline reads input. If `NULL`, Readline defaults to *stdin*.

`FILE * rl_outstream` [Variable]
The stdio stream to which Readline performs output. If `NULL`, Readline defaults to *stdout*.

`int rl_prefer_env_winsize` [Variable]

If non-zero, Readline gives values found in the `LINES` and `COLUMNS` environment variables greater precedence than values fetched from the kernel when computing the screen dimensions.

`rl_command_func_t * rl_last_func` [Variable]

The address of the last command function Readline executed. May be used to test whether or not a function is being executed twice in succession, for example.

`rl_hook_func_t * rl_startup_hook` [Variable]

If non-zero, this is the address of a function to call just before `readline` prints the first prompt.

`rl_hook_func_t * rl_pre_input_hook` [Variable]

If non-zero, this is the address of a function to call after the first prompt has been printed and just before `readline` starts reading input characters.

`rl_hook_func_t * rl_event_hook` [Variable]

If non-zero, this is the address of a function to call periodically when Readline is waiting for terminal input. By default, this will be called at most ten times a second if there is no keyboard input.

`rl_getc_func_t * rl_getc_function` [Variable]

If non-zero, Readline will call indirectly through this pointer to get a character from the input stream. By default, it is set to `rl_getc`, the default Readline character input function (see Section 2.4.8 [Character Input], page 39). In general, an application that sets *rl_getc_function* should consider setting *rl_input_available_hook* as well.

`rl_hook_func_t * rl_signal_event_hook` [Variable]

If non-zero, this is the address of a function to call if a read system call is interrupted when Readline is reading terminal input.

`rl_hook_func_t * rl_input_available_hook` [Variable]

If non-zero, Readline will use this function's return value when it needs to determine whether or not there is available input on the current input source. The default hook checks `rl_instream`; if an application is using a different input source, it should set the hook appropriately. Readline queries for available input when implementing intra-key-sequence timeouts during input and incremental searches. This may use an application-specific timeout before returning a value; Readline uses the value passed to `rl_set_keyboard_input_timeout()` or the value of the user-settable *keyseq-timeout* variable. This is designed for use by applications using Readline's callback interface (see Section 2.4.12 [Alternate Interface], page 43), which may not use the traditional `read(2)` and file descriptor interface, or other applications using a different input mechanism. If an application uses an input mechanism or hook that can potentially exceed the value of *keyseq-timeout*, it should increase the timeout or set this hook appropriately even when not using the callback interface. In general, an application that sets *rl_getc_function* should consider setting *rl_input_available_hook* as well.

`rl_voidfunc_t * rl_redisplay_function` [Variable]

If non-zero, Readline will call indirectly through this pointer to update the display with the current contents of the editing buffer. By default, it is set to `rl_redisplay`, the default Readline redisplay function (see Section 2.4.6 [Redisplay], page 37).

`rl_vintfunc_t * rl_prep_term_function` [Variable]

If non-zero, Readline will call indirectly through this pointer to initialize the terminal. The function takes a single argument, an `int` flag that says whether or not to use eight-bit characters. By default, this is set to `rl_prep_terminal` (see Section 2.4.9 [Terminal Management], page 40).

`rl_voidfunc_t * rl_deprep_term_function` [Variable]

If non-zero, Readline will call indirectly through this pointer to reset the terminal. This function should undo the effects of `rl_prep_term_function`. By default, this is set to `rl_deprep_terminal` (see Section 2.4.9 [Terminal Management], page 40).

`Keymap rl_executing_keymap` [Variable]

This variable is set to the keymap (see Section 2.4.2 [Keymaps], page 33) in which the currently executing readline function was found.

`Keymap rl_binding_keymap` [Variable]

This variable is set to the keymap (see Section 2.4.2 [Keymaps], page 33) in which the last key binding occurred.

`char * rl_executing_macro` [Variable]

This variable is set to the text of any currently-executing macro.

`int rl_executing_key` [Variable]

The key that caused the dispatch to the currently-executing Readline function.

`char * rl_executing_keyseq` [Variable]

The full key sequence that caused the dispatch to the currently-executing Readline function.

`int rl_key_sequence_length` [Variable]

The number of characters in *rl_executing_keyseq*.

`int rl_readline_state` [Variable]

A variable with bit values that encapsulate the current Readline state. A bit is set with the `RL_SETSTATE` macro, and unset with the `RL_UNSETSTATE` macro. Use the `RL_ISSTATE` macro to test whether a particular state bit is set. Current state bits include:

`RL_STATE_NONE`
: Readline has not yet been called, nor has it begun to initialize.

`RL_STATE_INITIALIZING`
: Readline is initializing its internal data structures.

`RL_STATE_INITIALIZED`
: Readline has completed its initialization.

RL_STATE_TERMPREPPED
: Readline has modified the terminal modes to do its own input and redisplay.

RL_STATE_READCMD
: Readline is reading a command from the keyboard.

RL_STATE_METANEXT
: Readline is reading more input after reading the meta-prefix character.

RL_STATE_DISPATCHING
: Readline is dispatching to a command.

RL_STATE_MOREINPUT
: Readline is reading more input while executing an editing command.

RL_STATE_ISEARCH
: Readline is performing an incremental history search.

RL_STATE_NSEARCH
: Readline is performing a non-incremental history search.

RL_STATE_SEARCH
: Readline is searching backward or forward through the history for a string.

RL_STATE_NUMERICARG
: Readline is reading a numeric argument.

RL_STATE_MACROINPUT
: Readline is currently getting its input from a previously-defined keyboard macro.

RL_STATE_MACRODEF
: Readline is currently reading characters defining a keyboard macro.

RL_STATE_OVERWRITE
: Readline is in overwrite mode.

RL_STATE_COMPLETING
: Readline is performing word completion.

RL_STATE_SIGHANDLER
: Readline is currently executing the readline signal handler.

RL_STATE_UNDOING
: Readline is performing an undo.

RL_STATE_INPUTPENDING
: Readline has input pending due to a call to `rl_execute_next()`.

RL_STATE_TTYCSAVED
: Readline has saved the values of the terminal's special characters.

RL_STATE_CALLBACK
: Readline is currently using the alternate (callback) interface (see Section 2.4.12 [Alternate Interface], page 43).

`RL_STATE_VIMOTION`
: Readline is reading the argument to a vi-mode "motion" command.

`RL_STATE_MULTIKEY`
: Readline is reading a multiple-keystroke command.

`RL_STATE_VICMDONCE`
: Readline has entered vi command (movement) mode at least one time during the current call to `readline()`.

`RL_STATE_DONE`
: Readline has read a key sequence bound to `accept-line` and is about to return the line to the caller.

`int rl_explicit_arg` [Variable]

Set to a non-zero value if an explicit numeric argument was specified by the user. Only valid in a bindable command function.

`int rl_numeric_arg` [Variable]

Set to the value of any numeric argument explicitly specified by the user before executing the current Readline function. Only valid in a bindable command function.

`int rl_editing_mode` [Variable]

Set to a value denoting Readline's current editing mode. A value of *1* means Readline is currently in emacs mode; *0* means that vi mode is active.

2.4 Readline Convenience Functions

2.4.1 Naming a Function

The user can dynamically change the bindings of keys while using Readline. This is done by representing the function with a descriptive name. The user is able to type the descriptive name when referring to the function. Thus, in an init file, one might find

```
Meta-Rubout:  backward-kill-word
```

This binds the keystroke `Meta-Rubout` to the function *descriptively* named `backward-kill-word`. You, as the programmer, should bind the functions you write to descriptive names as well. Readline provides a function for doing that:

`int rl_add_defun` (*const char *name, rl_command_func_t *function, int key*) [Function]

Add *name* to the list of named functions. Make *function* be the function that gets called. If *key* is not -1, then bind it to *function* using `rl_bind_key()`.

Using this function alone is sufficient for most applications. It is the recommended way to add a few functions to the default functions that Readline has built in. If you need to do something other than adding a function to Readline, you may need to use the underlying functions described below.

2.4.2 Selecting a Keymap

Key bindings take place on a *keymap*. The keymap is the association between the keys that the user types and the functions that get run. You can make your own keymaps, copy existing keymaps, and tell Readline which keymap to use.

`Keymap rl_make_bare_keymap` (*void*) [Function]
: Returns a new, empty keymap. The space for the keymap is allocated with `malloc()`; the caller should free it by calling `rl_free_keymap()` when done.

`Keymap rl_copy_keymap` (*Keymap map*) [Function]
: Return a new keymap which is a copy of *map*.

`Keymap rl_make_keymap` (*void*) [Function]
: Return a new keymap with the printing characters bound to rl_insert, the lowercase Meta characters bound to run their equivalents, and the Meta digits bound to produce numeric arguments.

`void rl_discard_keymap` (*Keymap keymap*) [Function]
: Free the storage associated with the data in *keymap*. The caller should free *keymap*.

`void rl_free_keymap` (*Keymap keymap*) [Function]
: Free all storage associated with *keymap*. This calls `rl_discard_keymap` to free subordindate keymaps and macros.

`int rl_empty_keymap` (*Keymap keymap*) [Function]
: Return non-zero if there are no keys bound to functions in *keymap*; zero if there are any keys bound.

Readline has several internal keymaps. These functions allow you to change which keymap is active.

`Keymap rl_get_keymap` (*void*) [Function]
: Returns the currently active keymap.

`void rl_set_keymap` (*Keymap keymap*) [Function]
: Makes *keymap* the currently active keymap.

`Keymap rl_get_keymap_by_name` (*const char *name*) [Function]
: Return the keymap matching *name*. *name* is one which would be supplied in a `set keymap` inputrc line (see Section 1.3 [Readline Init File], page 4).

`char * rl_get_keymap_name` (*Keymap keymap*) [Function]
: Return the name matching *keymap*. *name* is one which would be supplied in a `set keymap` inputrc line (see Section 1.3 [Readline Init File], page 4).

`int rl_set_keymap_name` (*const char *name, Keymap keymap*) [Function]
: Set the name of *keymap*. This name will then be "registered" and available for use in a `set keymap` inputrc directive see Section 1.3 [Readline Init File], page 4). The *name* may not be one of Readline's builtin keymap names; you may not add a different name for one of Readline's builtin keymaps. You may replace the name associated

with a given keymap by calling this function more than once with the same *keymap* argument. You may associate a registered *name* with a new keymap by calling this function more than once with the same *name* argument. There is no way to remove a named keymap once the name has been registered. Readline will make a copy of *name*. The return value is greater than zero unless *name* is one of Readline's builtin keymap names or *keymap* is one of Readline's builtin keymaps.

2.4.3 Binding Keys

Key sequences are associate with functions through the keymap. Readline has several internal keymaps: `emacs_standard_keymap`, `emacs_meta_keymap`, `emacs_ctlx_keymap`, `vi_movement_keymap`, and `vi_insertion_keymap`. `emacs_standard_keymap` is the default, and the examples in this manual assume that.

Since `readline()` installs a set of default key bindings the first time it is called, there is always the danger that a custom binding installed before the first call to `readline()` will be overridden. An alternate mechanism is to install custom key bindings in an initialization function assigned to the `rl_startup_hook` variable (see Section 2.3 [Readline Variables], page 27).

These functions manage key bindings.

`int rl_bind_key` (*int key, rl_command_func_t *function*) [Function]

Binds *key* to *function* in the currently active keymap. Returns non-zero in the case of an invalid *key*.

`int rl_bind_key_in_map` (*int key, rl_command_func_t *function, Keymap map*) [Function]

Bind *key* to *function* in *map*. Returns non-zero in the case of an invalid *key*.

`int rl_bind_key_if_unbound` (*int key, rl_command_func_t *function*) [Function]

Binds *key* to *function* if it is not already bound in the currently active keymap. Returns non-zero in the case of an invalid *key* or if *key* is already bound.

`int rl_bind_key_if_unbound_in_map` (*int key, rl_command_func_t *function, Keymap map*) [Function]

Binds *key* to *function* if it is not already bound in *map*. Returns non-zero in the case of an invalid *key* or if *key* is already bound.

`int rl_unbind_key` (*int key*) [Function]

Bind *key* to the null function in the currently active keymap. Returns non-zero in case of error.

`int rl_unbind_key_in_map` (*int key, Keymap map*) [Function]

Bind *key* to the null function in *map*. Returns non-zero in case of error.

`int rl_unbind_function_in_map` (*rl_command_func_t *function, Keymap map*) [Function]

Unbind all keys that execute *function* in *map*.

`int rl_unbind_command_in_map` (*const char *command, Keymap map*) [Function]

Unbind all keys that are bound to *command* in *map*.

`int rl_bind_keyseq` (*const char *keyseq, rl_command_func_t *function*) [Function]

Bind the key sequence represented by the string *keyseq* to the function *function*, beginning in the current keymap. This makes new keymaps as necessary. The return value is non-zero if *keyseq* is invalid.

`int rl_bind_keyseq_in_map` (*const char *keyseq, rl_command_func_t *function, Keymap map*) [Function]

Bind the key sequence represented by the string *keyseq* to the function *function*. This makes new keymaps as necessary. Initial bindings are performed in *map*. The return value is non-zero if *keyseq* is invalid.

`int rl_set_key` (*const char *keyseq, rl_command_func_t *function, Keymap map*) [Function]

Equivalent to `rl_bind_keyseq_in_map`.

`int rl_bind_keyseq_if_unbound` (*const char *keyseq, rl_command_func_t *function*) [Function]

Binds *keyseq* to *function* if it is not already bound in the currently active keymap. Returns non-zero in the case of an invalid *keyseq* or if *keyseq* is already bound.

`int rl_bind_keyseq_if_unbound_in_map` (*const char *keyseq, rl_command_func_t *function, Keymap map*) [Function]

Binds *keyseq* to *function* if it is not already bound in *map*. Returns non-zero in the case of an invalid *keyseq* or if *keyseq* is already bound.

`int rl_generic_bind` (*int type, const char *keyseq, char *data, Keymap map*) [Function]

Bind the key sequence represented by the string *keyseq* to the arbitrary pointer *data*. *type* says what kind of data is pointed to by *data*; this can be a function (`ISFUNC`), a macro (`ISMACR`), or a keymap (`ISKMAP`). This makes new keymaps as necessary. The initial keymap in which to do bindings is *map*.

`int rl_parse_and_bind` (*char *line*) [Function]

Parse *line* as if it had been read from the `inputrc` file and perform any key bindings and variable assignments found (see Section 1.3 [Readline Init File], page 4).

`int rl_read_init_file` (*const char *filename*) [Function]

Read keybindings and variable assignments from *filename* (see Section 1.3 [Readline Init File], page 4).

2.4.4 Associating Function Names and Bindings

These functions allow you to find out what keys invoke named functions and the functions invoked by a particular key sequence. You may also associate a new function name with an arbitrary function.

`rl_command_func_t * rl_named_function` (*const char *name*) [Function]
Return the function with name *name*.

`rl_command_func_t * rl_function_of_keyseq` (*const char *keyseq, Keymap map, int *type*) [Function]
Return the function invoked by *keyseq* in keymap *map*. If *map* is `NULL`, the current keymap is used. If *type* is not `NULL`, the type of the object is returned in the `int` variable it points to (one of `ISFUNC`, `ISKMAP`, or `ISMACR`). It takes a "translated" key sequence and should not be used if the key sequence can include NUL.

`rl_command_func_t * rl_function_of_keyseq_len` (*const char *keyseq, size_t len, Keymap map, int *type*) [Function]
Return the function invoked by *keyseq* of length *len* in keymap *map*. Equivalent to `rl_function_of_keyseq` with the addition of the *len* parameter. It takes a "translated" key sequence and should be used if the key sequence can include NUL.

`char ** rl_invoking_keyseqs` (*rl_command_func_t *function*) [Function]
Return an array of strings representing the key sequences used to invoke *function* in the current keymap.

`char ** rl_invoking_keyseqs_in_map` (*rl_command_func_t *function, Keymap map*) [Function]
Return an array of strings representing the key sequences used to invoke *function* in the keymap *map*.

`void rl_function_dumper` (*int readable*) [Function]
Print the readline function names and the key sequences currently bound to them to `rl_outstream`. If *readable* is non-zero, the list is formatted in such a way that it can be made part of an `inputrc` file and re-read.

`void rl_list_funmap_names` (*void*) [Function]
Print the names of all bindable Readline functions to `rl_outstream`.

`const char ** rl_funmap_names` (*void*) [Function]
Return a NULL terminated array of known function names. The array is sorted. The array itself is allocated, but not the strings inside. You should free the array, but not the pointers, using `free` or `rl_free` when you are done.

`int rl_add_funmap_entry` (*const char *name, rl_command_func_t *function*) [Function]
Add *name* to the list of bindable Readline command names, and make *function* the function to be called when *name* is invoked.

2.4.5 Allowing Undoing

Supporting the undo command is a painless thing, and makes your functions much more useful. It is certainly easy to try something if you know you can undo it.

If your function simply inserts text once, or deletes text once, and uses `rl_insert_text()` or `rl_delete_text()` to do it, then undoing is already done for you automatically.

If you do multiple insertions or multiple deletions, or any combination of these operations, you should group them together into one operation. This is done with `rl_begin_undo_group()` and `rl_end_undo_group()`.

The types of events that can be undone are:

```
enum undo_code { UNDO_DELETE, UNDO_INSERT, UNDO_BEGIN, UNDO_END };
```

Notice that `UNDO_DELETE` means to insert some text, and `UNDO_INSERT` means to delete some text. That is, the undo code tells what to undo, not how to undo it. `UNDO_BEGIN` and `UNDO_END` are tags added by `rl_begin_undo_group()` and `rl_end_undo_group()`.

`int rl_begin_undo_group` (*void*) [Function]

Begins saving undo information in a group construct. The undo information usually comes from calls to `rl_insert_text()` and `rl_delete_text()`, but could be the result of calls to `rl_add_undo()`.

`int rl_end_undo_group` (*void*) [Function]

Closes the current undo group started with `rl_begin_undo_group ()`. There should be one call to `rl_end_undo_group()` for each call to `rl_begin_undo_group()`.

`void rl_add_undo` (*enum undo_code what, int start, int end, char *text*) [Function]

Remember how to undo an event (according to *what*). The affected text runs from *start* to *end*, and encompasses *text*.

`void rl_free_undo_list` (*void*) [Function]

Free the existing undo list.

`int rl_do_undo` (*void*) [Function]

Undo the first thing on the undo list. Returns `0` if there was nothing to undo, non-zero if something was undone.

Finally, if you neither insert nor delete text, but directly modify the existing text (e.g., change its case), call `rl_modifying()` once, just before you modify the text. You must supply the indices of the text range that you are going to modify.

`int rl_modifying` (*int start, int end*) [Function]

Tell Readline to save the text between *start* and *end* as a single undo unit. It is assumed that you will subsequently modify that text.

2.4.6 Redisplay

`void rl_redisplay` (*void*) [Function]

Change what's displayed on the screen to reflect the current contents of `rl_line_buffer`.

`int rl_forced_update_display` (*void*) [Function]

Force the line to be updated and redisplayed, whether or not Readline thinks the screen display is correct.

`int rl_on_new_line` (*void*) [Function]

Tell the update functions that we have moved onto a new (empty) line, usually after outputting a newline.

`int rl_on_new_line_with_prompt` (*void*) [Function]
: Tell the update functions that we have moved onto a new line, with *rl_prompt* already displayed. This could be used by applications that want to output the prompt string themselves, but still need Readline to know the prompt string length for redisplay. It should be used after setting *rl_already_prompted*.

`int rl_clear_visible_line` (*void*) [Function]
: Clear the screen lines corresponding to the current line's contents.

`int rl_reset_line_state` (*void*) [Function]
: Reset the display state to a clean state and redisplay the current line starting on a new line.

`int rl_crlf` (*void*) [Function]
: Move the cursor to the start of the next screen line.

`int rl_show_char` (*int c*) [Function]
: Display character *c* on `rl_outstream`. If Readline has not been set to display meta characters directly, this will convert meta characters to a meta-prefixed key sequence. This is intended for use by applications which wish to do their own redisplay.

`int rl_message` (*const char *, . . .*) [Function]
: The arguments are a format string as would be supplied to `printf`, possibly containing conversion specifications such as '`%d`', and any additional arguments necessary to satisfy the conversion specifications. The resulting string is displayed in the *echo area*. The echo area is also used to display numeric arguments and search strings. You should call `rl_save_prompt` to save the prompt information before calling this function.

`int rl_clear_message` (*void*) [Function]
: Clear the message in the echo area. If the prompt was saved with a call to `rl_save_prompt` before the last call to `rl_message`, call `rl_restore_prompt` before calling this function.

`void rl_save_prompt` (*void*) [Function]
: Save the local Readline prompt display state in preparation for displaying a new message in the message area with `rl_message()`.

`void rl_restore_prompt` (*void*) [Function]
: Restore the local Readline prompt display state saved by the most recent call to `rl_save_prompt`. if `rl_save_prompt` was called to save the prompt before a call to `rl_message`, this function should be called before the corresponding call to `rl_clear_message`.

`int rl_expand_prompt` (*char *prompt*) [Function]
: Expand any special character sequences in *prompt* and set up the local Readline prompt redisplay variables. This function is called by `readline()`. It may also be called to expand the primary prompt if the `rl_on_new_line_with_prompt()` function or `rl_already_prompted` variable is used. It returns the number of visible characters on the last line of the (possibly multi-line) prompt. Applications may indicate that

the prompt contains characters that take up no physical screen space when displayed by bracketing a sequence of such characters with the special markers `RL_PROMPT_START_IGNORE` and `RL_PROMPT_END_IGNORE` (declared in `readline.h`). This may be used to embed terminal-specific escape sequences in prompts.

`int rl_set_prompt` (*const char *prompt*) [Function]
Make Readline use *prompt* for subsequent redisplay. This calls `rl_expand_prompt()` to expand the prompt and sets `rl_prompt` to the result.

2.4.7 Modifying Text

`int rl_insert_text` (*const char *text*) [Function]
Insert *text* into the line at the current cursor position. Returns the number of characters inserted.

`int rl_delete_text` (*int start, int end*) [Function]
Delete the text between *start* and *end* in the current line. Returns the number of characters deleted.

`char * rl_copy_text` (*int start, int end*) [Function]
Return a copy of the text between *start* and *end* in the current line.

`int rl_kill_text` (*int start, int end*) [Function]
Copy the text between *start* and *end* in the current line to the kill ring, appending or prepending to the last kill if the last command was a kill command. The text is deleted. If *start* is less than *end*, the text is appended, otherwise prepended. If the last command was not a kill, a new kill ring slot is used.

`int rl_push_macro_input` (*char *macro*) [Function]
Cause *macro* to be inserted into the line, as if it had been invoked by a key bound to a macro. Not especially useful; use `rl_insert_text()` instead.

2.4.8 Character Input

`int rl_read_key` (*void*) [Function]
Return the next character available from Readline's current input stream. This handles input inserted into the input stream via *rl_pending_input* (see Section 2.3 [Readline Variables], page 27) and `rl_stuff_char()`, macros, and characters read from the keyboard. While waiting for input, this function will call any function assigned to the `rl_event_hook` variable.

`int rl_getc` (*FILE *stream*) [Function]
Return the next character available from *stream*, which is assumed to be the keyboard.

`int rl_stuff_char` (*int c*) [Function]
Insert *c* into the Readline input stream. It will be "read" before Readline attempts to read characters from the terminal with `rl_read_key()`. Up to 512 characters may be pushed back. `rl_stuff_char` returns 1 if the character was successfully inserted; 0 otherwise.

`int rl_execute_next` (*int c*) [Function]
Make *c* be the next command to be executed when `rl_read_key()` is called. This sets *rl_pending_input*.

`int rl_clear_pending_input` (*void*) [Function]
Unset *rl_pending_input*, effectively negating the effect of any previous call to `rl_execute_next()`. This works only if the pending input has not already been read with `rl_read_key()`.

`int rl_set_keyboard_input_timeout` (*int u*) [Function]
While waiting for keyboard input in `rl_read_key()`, Readline will wait for *u* microseconds for input before calling any function assigned to `rl_event_hook`. *u* must be greater than or equal to zero (a zero-length timeout is equivalent to a poll). The default waiting period is one-tenth of a second. Returns the old timeout value.

2.4.9 Terminal Management

`void rl_prep_terminal` (*int meta_flag*) [Function]
Modify the terminal settings for Readline's use, so `readline()` can read a single character at a time from the keyboard. The *meta_flag* argument should be non-zero if Readline should read eight-bit input.

`void rl_deprep_terminal` (*void*) [Function]
Undo the effects of `rl_prep_terminal()`, leaving the terminal in the state in which it was before the most recent call to `rl_prep_terminal()`.

`void rl_tty_set_default_bindings` (*Keymap kmap*) [Function]
Read the operating system's terminal editing characters (as would be displayed by `stty`) to their Readline equivalents. The bindings are performed in *kmap*.

`void rl_tty_unset_default_bindings` (*Keymap kmap*) [Function]
Reset the bindings manipulated by `rl_tty_set_default_bindings` so that the terminal editing characters are bound to `rl_insert`. The bindings are performed in *kmap*.

`int rl_tty_set_echoing` (*int value*) [Function]
Set Readline's idea of whether or not it is echoing output to its output stream (*rl_outstream*). If *value* is 0, Readline does not display output to *rl_outstream*; any other value enables output. The initial value is set when Readline initializes the terminal settings. This function returns the previous value.

`int rl_reset_terminal` (*const char *terminal_name*) [Function]
Reinitialize Readline's idea of the terminal settings using *terminal_name* as the terminal type (e.g., `vt100`). If *terminal_name* is `NULL`, the value of the `TERM` environment variable is used.

2.4.10 Utility Functions

`int rl_save_state` (*struct readline_state *sp*) [Function]
Save a snapshot of Readline's internal state to *sp*. The contents of the *readline_state* structure are documented in `readline.h`. The caller is responsible for allocating the structure.

`int rl_restore_state` (*struct readline_state *sp*) [Function]

Restore Readline's internal state to that stored in *sp*, which must have been saved by a call to `rl_save_state`. The contents of the *readline_state* structure are documented in `readline.h`. The caller is responsible for freeing the structure.

`void rl_free` (*void *mem*) [Function]

Deallocate the memory pointed to by *mem*. *mem* must have been allocated by `malloc`.

`void rl_replace_line` (*const char *text, int clear_undo*) [Function]

Replace the contents of `rl_line_buffer` with *text*. The point and mark are preserved, if possible. If *clear_undo* is non-zero, the undo list associated with the current line is cleared.

`void rl_extend_line_buffer` (*int len*) [Function]

Ensure that `rl_line_buffer` has enough space to hold *len* characters, possibly reallocating it if necessary.

`int rl_initialize` (*void*) [Function]

Initialize or re-initialize Readline's internal state. It's not strictly necessary to call this; `readline()` calls it before reading any input.

`int rl_ding` (*void*) [Function]

Ring the terminal bell, obeying the setting of `bell-style`.

`int rl_alphabetic` (*int c*) [Function]

Return 1 if *c* is an alphabetic character.

`void rl_display_match_list` (*char **matches, int len, int max*) [Function]

A convenience function for displaying a list of strings in columnar format on Readline's output stream. `matches` is the list of strings, in argv format, such as a list of completion matches. `len` is the number of strings in `matches`, and `max` is the length of the longest string in `matches`. This function uses the setting of `print-completions-horizontally` to select how the matches are displayed (see Section 1.3.1 [Readline Init File Syntax], page 4). When displaying completions, this function sets the number of columns used for display to the value of `completion-display-width`, the value of the environment variable `COLUMNS`, or the screen width, in that order.

The following are implemented as macros, defined in `chardefs.h`. Applications should refrain from using them.

`int _rl_uppercase_p` (*int c*) [Function]

Return 1 if *c* is an uppercase alphabetic character.

`int _rl_lowercase_p` (*int c*) [Function]

Return 1 if *c* is a lowercase alphabetic character.

`int _rl_digit_p` (*int c*) [Function]

Return 1 if *c* is a numeric character.

`int _rl_to_upper` (*int c*) [Function]

If *c* is a lowercase alphabetic character, return the corresponding uppercase character.

`int _rl_to_lower` (*int c*) [Function]
If *c* is an uppercase alphabetic character, return the corresponding lowercase character.

`int _rl_digit_value` (*int c*) [Function]
If *c* is a number, return the value it represents.

2.4.11 Miscellaneous Functions

`int rl_macro_bind` (*const char *keyseq, const char *macro, Keymap map*) [Function]
Bind the key sequence *keyseq* to invoke the macro *macro*. The binding is performed in *map*. When *keyseq* is invoked, the *macro* will be inserted into the line. This function is deprecated; use `rl_generic_bind()` instead.

`void rl_macro_dumper` (*int readable*) [Function]
Print the key sequences bound to macros and their values, using the current keymap, to `rl_outstream`. If *readable* is non-zero, the list is formatted in such a way that it can be made part of an `inputrc` file and re-read.

`int rl_variable_bind` (*const char *variable, const char *value*) [Function]
Make the Readline variable *variable* have *value*. This behaves as if the readline command '`set` *`variable value`*' had been executed in an `inputrc` file (see Section 1.3.1 [Readline Init File Syntax], page 4).

`char * rl_variable_value` (*const char *variable*) [Function]
Return a string representing the value of the Readline variable *variable*. For boolean variables, this string is either '`on`' or '`off`'.

`void rl_variable_dumper` (*int readable*) [Function]
Print the readline variable names and their current values to `rl_outstream`. If *readable* is non-zero, the list is formatted in such a way that it can be made part of an `inputrc` file and re-read.

`int rl_set_paren_blink_timeout` (*int u*) [Function]
Set the time interval (in microseconds) that Readline waits when showing a balancing character when `blink-matching-paren` has been enabled.

`char * rl_get_termcap` (*const char *cap*) [Function]
Retrieve the string value of the termcap capability *cap*. Readline fetches the termcap entry for the current terminal name and uses those capabilities to move around the screen line and perform other terminal-specific operations, like erasing a line. Readline does not use all of a terminal's capabilities, and this function will return values for only those capabilities Readline uses.

`void rl_clear_history` (*void*) [Function]
Clear the history list by deleting all of the entries, in the same manner as the History library's `clear_history()` function. This differs from `clear_history` because it frees private data Readline saves in the history list.

2.4.12 Alternate Interface

An alternate interface is available to plain `readline()`. Some applications need to interleave keyboard I/O with file, device, or window system I/O, typically by using a main loop to `select()` on various file descriptors. To accommodate this need, readline can also be invoked as a 'callback' function from an event loop. There are functions available to make this easy.

`void rl_callback_handler_install` (*const char *prompt, rl_vcpfunc_t *lhandler*) [Function]

Set up the terminal for readline I/O and display the initial expanded value of *prompt*. Save the value of *lhandler* to use as a handler function to call when a complete line of input has been entered. The handler function receives the text of the line as an argument. As with `readline()`, the handler function should `free` the line when it it finished with it.

`void rl_callback_read_char` (*void*) [Function]

Whenever an application determines that keyboard input is available, it should call `rl_callback_read_char()`, which will read the next character from the current input source. If that character completes the line, `rl_callback_read_char` will invoke the *lhandler* function installed by `rl_callback_handler_install` to process the line. Before calling the *lhandler* function, the terminal settings are reset to the values they had before calling `rl_callback_handler_install`. If the *lhandler* function returns, and the line handler remains installed, the terminal settings are modified for Readline's use again. `EOF` is indicated by calling *lhandler* with a `NULL` line.

`void rl_callback_sigcleanup` (*void*) [Function]

Clean up any internal state the callback interface uses to maintain state between calls to rl_callback_read_char (e.g., the state of any active incremental searches). This is intended to be used by applications that wish to perform their own signal handling; Readline's internal signal handler calls this when appropriate.

`void rl_callback_handler_remove` (*void*) [Function]

Restore the terminal to its initial state and remove the line handler. You may call this function from within a callback as well as independently. If the *lhandler* installed by `rl_callback_handler_install` does not exit the program, either this function or the function referred to by the value of `rl_deprep_term_function` should be called before the program exits to reset the terminal settings.

2.4.13 A Readline Example

Here is a function which changes lowercase characters to their uppercase equivalents, and uppercase characters to lowercase. If this function was bound to '`M-c`', then typing '`M-c`' would change the case of the character under point. Typing '`M-1 0 M-c`' would change the case of the following 10 characters, leaving the cursor on the last character changed.

```
/* Invert the case of the COUNT following characters. */
int
invert_case_line (count, key)
     int count, key;
```

```
{
  register int start, end, i;

  start = rl_point;

  if (rl_point >= rl_end)
    return (0);

  if (count < 0)
    {
      direction = -1;
      count = -count;
    }
  else
    direction = 1;

  /* Find the end of the range to modify. */
  end = start + (count * direction);

  /* Force it to be within range. */
  if (end > rl_end)
    end = rl_end;
  else if (end < 0)
    end = 0;

  if (start == end)
    return (0);

  if (start > end)
    {
      int temp = start;
      start = end;
      end = temp;
    }

  /* Tell readline that we are modifying the line,
     so it will save the undo information. */
  rl_modifying (start, end);

  for (i = start; i != end; i++)
    {
      if (_rl_uppercase_p (rl_line_buffer[i]))
        rl_line_buffer[i] = _rl_to_lower (rl_line_buffer[i]);
      else if (_rl_lowercase_p (rl_line_buffer[i]))
        rl_line_buffer[i] = _rl_to_upper (rl_line_buffer[i]);
    }
  /* Move point to on top of the last character changed. */
```

```
  rl_point = (direction == 1) ? end - 1 : start;
  return (0);
}
```

2.4.14 Alternate Interface Example

Here is a complete program that illustrates Readline's alternate interface. It reads lines from the terminal and displays them, providing the standard history and TAB completion functions. It understands the EOF character or "exit" to exit the program.

```
/* Standard include files. stdio.h is required. */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <locale.h>

/* Used for select(2) */
#include <sys/types.h>
#include <sys/select.h>

#include <signal.h>

#include <stdio.h>

/* Standard readline include files. */
#include <readline/readline.h>
#include <readline/history.h>

static void cb_linehandler (char *);
static void sighandler (int);

int running;
int sigwinch_received;
const char *prompt = "rltest$ ";

/* Handle SIGWINCH and window size changes when readline is not active and
   reading a character. */
static void
sighandler (int sig)
{
  sigwinch_received = 1;
}

/* Callback function called for each line when accept-line executed, EOF
   seen, or EOF character read.  This sets a flag and returns; it could
   also call exit(3). */
static void
cb_linehandler (char *line)
```

```
{
  /* Can use ^D (stty eof) or ‘exit’ to exit. */
  if (line == NULL || strcmp (line, "exit") == 0)
    {
      if (line == 0)
        printf ("\n");
      printf ("exit\n");
      /* This function needs to be called to reset the terminal settings,
         and calling it from the line handler keeps one extra prompt from
         being displayed. */
      rl_callback_handler_remove ();

      running = 0;
    }
  else
    {
      if (*line)
        add_history (line);
      printf ("input line: %s\n", line);
      free (line);
    }
}

int
main (int c, char **v)
{
  fd_set fds;
  int r;

  /* Set the default locale values according to environment variables. */
  setlocale (LC_ALL, "");

  /* Handle window size changes when readline is not active and reading
     characters. */
  signal (SIGWINCH, sighandler);

  /* Install the line handler. */
  rl_callback_handler_install (prompt, cb_linehandler);

  /* Enter a simple event loop.  This waits until something is available
     to read on readline’s input stream (defaults to standard input) and
     calls the builtin character read callback to read it.  It does not
     have to modify the user’s terminal settings. */
  running = 1;
  while (running)
    {
      FD_ZERO (&fds);
```

```
          FD_SET (fileno (rl_instream), &fds);

          r = select (FD_SETSIZE, &fds, NULL, NULL, NULL);
          if (r < 0 && errno != EINTR)
            {
              perror ("rltest: select");
              rl_callback_handler_remove ();
              break;
            }
          if (sigwinch_received)
    {
      rl_resize_terminal ();
      sigwinch_received = 0;
    }
          if (r < 0)
    continue;

          if (FD_ISSET (fileno (rl_instream), &fds))
            rl_callback_read_char ();
        }

      printf ("rltest: Event loop has exited\n");
      return 0;
    }
```

2.5 Readline Signal Handling

Signals are asynchronous events sent to a process by the Unix kernel, sometimes on behalf of another process. They are intended to indicate exceptional events, like a user pressing the interrupt key on his terminal, or a network connection being broken. There is a class of signals that can be sent to the process currently reading input from the keyboard. Since Readline changes the terminal attributes when it is called, it needs to perform special processing when such a signal is received in order to restore the terminal to a sane state, or provide application writers with functions to do so manually.

Readline contains an internal signal handler that is installed for a number of signals (`SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGHUP`, `SIGALRM`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU`). When one of these signals is received, the signal handler will reset the terminal attributes to those that were in effect before `readline()` was called, reset the signal handling to what it was before `readline()` was called, and resend the signal to the calling application. If and when the calling application's signal handler returns, Readline will reinitialize the terminal and continue to accept input. When a `SIGINT` is received, the Readline signal handler performs some additional work, which will cause any partially-entered line to be aborted (see the description of `rl_free_line_state()` below).

There is an additional Readline signal handler, for `SIGWINCH`, which the kernel sends to a process whenever the terminal's size changes (for example, if a user resizes an `xterm`). The Readline `SIGWINCH` handler updates Readline's internal screen size information, and then calls any `SIGWINCH` signal handler the calling application has installed. Readline calls the application's `SIGWINCH` signal handler without resetting the terminal to its original state. If the application's signal handler does more than update its idea of the terminal size and return (for example, a `longjmp` back to a main processing loop), it *must* call `rl_cleanup_after_signal()` (described below), to restore the terminal state.

When an application is using the callback interface (see Section 2.4.12 [Alternate Interface], page 43), Readline installs signal handlers only for the duration of the call to `rl_callback_read_char`. Applications using the callback interface should be prepared to clean up Readline's state if they wish to handle the signal before the line handler completes and restores the terminal state.

If an application using the callback interface wishes to have Readline install its signal handlers at the time the application calls `rl_callback_handler_install` and remove them only when a complete line of input has been read, it should set the `rl_persistent_signal_handlers` variable to a non-zero value. This allows an application to defer all of the handling of the signals Readline catches to Readline. Applications should use this variable with care; it can result in Readline catching signals and not acting on them (or allowing the application to react to them) until the application calls `rl_callback_read_char`. This can result in an application becoming less responsive to keyboard signals like SIGINT. If an application does not want or need to perform any signal handling, or does not need to do any processing between calls to `rl_callback_read_char`, setting this variable may be desirable.

Readline provides two variables that allow application writers to control whether or not it will catch certain signals and act on them when they are received. It is important that applications change the values of these variables only when calling `readline()`, not in a signal handler, so Readline's internal signal state is not corrupted.

`int rl_catch_signals` [Variable]

If this variable is non-zero, Readline will install signal handlers for `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGHUP`, `SIGALRM`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU`.

The default value of `rl_catch_signals` is 1.

`int rl_catch_sigwinch` [Variable]

If this variable is set to a non-zero value, Readline will install a signal handler for `SIGWINCH`.

The default value of `rl_catch_sigwinch` is 1.

`int rl_persistent_signal_handlers` [Variable]

If an application using the callback interface wishes Readline's signal handlers to be installed and active during the set of calls to `rl_callback_read_char` that constitutes an entire single line, it should set this variable to a non-zero value.

The default value of `rl_persistent_signal_handlers` is 0.

`int rl_change_environment` [Variable]

If this variable is set to a non-zero value, and Readline is handling `SIGWINCH`, Readline will modify the *LINES* and *COLUMNS* environment variables upon receipt of a `SIGWINCH`

The default value of `rl_change_environment` is 1.

If an application does not wish to have Readline catch any signals, or to handle signals other than those Readline catches (`SIGHUP`, for example), Readline provides convenience functions to do the necessary terminal and internal state cleanup upon receipt of a signal.

`int rl_pending_signal` (*void*) [Function]
Return the signal number of the most recent signal Readline received but has not yet handled, or 0 if there is no pending signal.

`void rl_cleanup_after_signal` (*void*) [Function]
This function will reset the state of the terminal to what it was before `readline()` was called, and remove the Readline signal handlers for all signals, depending on the values of `rl_catch_signals` and `rl_catch_sigwinch`.

`void rl_free_line_state` (*void*) [Function]
This will free any partial state associated with the current input line (undo information, any partial history entry, any partially-entered keyboard macro, and any partially-entered numeric argument). This should be called before `rl_cleanup_after_signal()`. The Readline signal handler for `SIGINT` calls this to abort the current input line.

`void rl_reset_after_signal` (*void*) [Function]
This will reinitialize the terminal and reinstall any Readline signal handlers, depending on the values of `rl_catch_signals` and `rl_catch_sigwinch`.

If an application wants to force Readline to handle any signals that have arrived while it has been executing, `rl_check_signals()` will call Readline's internal signal handler if there are any pending signals. This is primarily intended for those applications that use a custom `rl_getc_function` (see Section 2.3 [Readline Variables], page 27) and wish to handle signals received while waiting for input.

`void rl_check_signals` (*void*) [Function]
If there are any pending signals, call Readline's internal signal handling functions to process them. `rl_pending_signal()` can be used independently to determine whether or not there are any pending signals.

If an application does not wish Readline to catch `SIGWINCH`, it may call `rl_resize_terminal()` or `rl_set_screen_size()` to force Readline to update its idea of the terminal size when a `SIGWINCH` is received.

`void rl_echo_signal_char` (*int sig*) [Function]
If an application wishes to install its own signal handlers, but still have readline display characters that generate signals, calling this function with *sig* set to `SIGINT`, `SIGQUIT`, or `SIGTSTP` will display the character generating that signal.

`void rl_resize_terminal` (*void*) [Function]
Update Readline's internal screen size by reading values from the kernel.

`void rl_set_screen_size` (*int rows, int cols*) [Function]
Set Readline's idea of the terminal size to *rows* rows and *cols* columns. If either *rows* or *columns* is less than or equal to 0, Readline's idea of that terminal dimension is unchanged.

If an application does not want to install a `SIGWINCH` handler, but is still interested in the screen dimensions, Readline's idea of the screen size may be queried.

`void rl_get_screen_size` (*int *rows, int *cols*) [Function]

Return Readline's idea of the terminal's size in the variables pointed to by the arguments.

`void rl_reset_screen_size` (*void*) [Function]

Cause Readline to reobtain the screen size and recalculate its dimensions.

The following functions install and remove Readline's signal handlers.

`int rl_set_signals` (*void*) [Function]

Install Readline's signal handler for `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGHUP`, `SIGALRM`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`, and `SIGWINCH`, depending on the values of `rl_catch_signals` and `rl_catch_sigwinch`.

`int rl_clear_signals` (*void*) [Function]

Remove all of the Readline signal handlers installed by `rl_set_signals()`.

2.6 Custom Completers

Typically, a program that reads commands from the user has a way of disambiguating commands and data. If your program is one of these, then it can provide completion for commands, data, or both. The following sections describe how your program and Readline cooperate to provide this service.

2.6.1 How Completing Works

In order to complete some text, the full list of possible completions must be available. That is, it is not possible to accurately expand a partial word without knowing all of the possible words which make sense in that context. The Readline library provides the user interface to completion, and two of the most common completion functions: filename and username. For completing other types of text, you must write your own completion function. This section describes exactly what such functions must do, and provides an example.

There are three major functions used to perform completion:

1. The user-interface function `rl_complete()`. This function is called with the same arguments as other bindable Readline functions: *count* and *invoking_key*. It isolates the word to be completed and calls `rl_completion_matches()` to generate a list of possible completions. It then either lists the possible completions, inserts the possible completions, or actually performs the completion, depending on which behavior is desired.
2. The internal function `rl_completion_matches()` uses an application-supplied *generator* function to generate the list of possible matches, and then returns the array of these matches. The caller should place the address of its generator function in `rl_completion_entry_function`.
3. The generator function is called repeatedly from `rl_completion_matches()`, returning a string each time. The arguments to the generator function are *text* and *state*. *text* is the partial word to be completed. *state* is zero the first time the function is called,

allowing the generator to perform any necessary initialization, and a positive non-zero integer for each subsequent call. The generator function returns `(char *)NULL` to inform `rl_completion_matches()` that there are no more possibilities left. Usually the generator function computes the list of possible completions when *state* is zero, and returns them one at a time on subsequent calls. Each string the generator function returns as a match must be allocated with `malloc()`; Readline frees the strings when it has finished with them. Such a generator function is referred to as an *application-specific completion function*.

`int rl_complete` (*int ignore, int invoking_key*) [Function]

Complete the word at or before point. You have supplied the function that does the initial simple matching selection algorithm (see `rl_completion_matches()`). The default is to do filename completion.

`rl_compentry_func_t * rl_completion_entry_function` [Variable]

This is a pointer to the generator function for `rl_completion_matches()`. If the value of `rl_completion_entry_function` is `NULL` then the default filename generator function, `rl_filename_completion_function()`, is used. An *application-specific completion function* is a function whose address is assigned to `rl_completion_entry_function` and whose return values are used to generate possible completions.

2.6.2 Completion Functions

Here is the complete list of callable completion functions present in Readline.

`int rl_complete_internal` (*int what_to_do*) [Function]

Complete the word at or before point. *what_to_do* says what to do with the completion. A value of '`?`' means list the possible completions. '`TAB`' means do standard completion. '`*`' means insert all of the possible completions. '`!`' means to display all of the possible completions, if there is more than one, as well as performing partial completion. '`@`' is similar to '`!`', but possible completions are not listed if the possible completions share a common prefix.

`int rl_complete` (*int ignore, int invoking_key*) [Function]

Complete the word at or before point. You have supplied the function that does the initial simple matching selection algorithm (see `rl_completion_matches()` and `rl_completion_entry_function`). The default is to do filename completion. This calls `rl_complete_internal()` with an argument depending on *invoking_key*.

`int rl_possible_completions` (*int count, int invoking_key*) [Function]

List the possible completions. See description of `rl_complete ()`. This calls `rl_complete_internal()` with an argument of '`?`'.

`int rl_insert_completions` (*int count, int invoking_key*) [Function]

Insert the list of possible completions into the line, deleting the partially-completed word. See description of `rl_complete()`. This calls `rl_complete_internal()` with an argument of '`*`'.

`int rl_completion_mode` (*rl_command_func_t *cfunc*) [Function]

Returns the appropriate value to pass to `rl_complete_internal()` depending on whether *cfunc* was called twice in succession and the values of the `show-all-if-`

`ambiguous` and `show-all-if-unmodified` variables. Application-specific completion functions may use this function to present the same interface as `rl_complete()`.

`char ** rl_completion_matches` (*const char *text, rl_compentry_func_t *entry_func*) [Function]

Returns an array of strings which is a list of completions for *text*. If there are no completions, returns `NULL`. The first entry in the returned array is the substitution for *text*. The remaining entries are the possible completions. The array is terminated with a `NULL` pointer.

entry_func is a function of two args, and returns a `char *`. The first argument is *text*. The second is a state argument; it is zero on the first call, and non-zero on subsequent calls. *entry_func* returns a `NULL` pointer to the caller when there are no more matches.

`char * rl_filename_completion_function` (*const char *text, int state*) [Function]

A generator function for filename completion in the general case. *text* is a partial filename. The Bash source is a useful reference for writing application-specific completion functions (the Bash completion functions call this and other Readline functions).

`char * rl_username_completion_function` (*const char *text, int state*) [Function]

A completion generator for usernames. *text* contains a partial username preceded by a random character (usually '~'). As with all completion generators, *state* is zero on the first call and non-zero for subsequent calls.

2.6.3 Completion Variables

`rl_compentry_func_t * rl_completion_entry_function` [Variable]

A pointer to the generator function for `rl_completion_matches()`. `NULL` means to use `rl_filename_completion_function()`, the default filename completer.

`rl_completion_func_t * rl_attempted_completion_function` [Variable]

A pointer to an alternative function to create matches. The function is called with *text*, *start*, and *end*. *start* and *end* are indices in `rl_line_buffer` defining the boundaries of *text*, which is a character string. If this function exists and returns `NULL`, or if this variable is set to `NULL`, then `rl_complete()` will call the value of `rl_completion_entry_function` to generate matches, otherwise the array of strings returned will be used. If this function sets the `rl_attempted_completion_over` variable to a non-zero value, Readline will not perform its default completion even if this function returns no matches.

`rl_quote_func_t * rl_filename_quoting_function` [Variable]

A pointer to a function that will quote a filename in an application-specific fashion. This is called if filename completion is being attempted and one of the characters in `rl_filename_quote_characters` appears in a completed filename. The function is called with *text*, *match_type*, and *quote_pointer*. The *text* is the filename to be quoted. The *match_type* is either `SINGLE_MATCH`, if there is only one completion match, or `MULT_MATCH`. Some functions use this to decide whether or not to insert a closing quote character. The *quote_pointer* is a pointer to any opening quote character the user typed. Some functions choose to reset this character.

`rl_dequote_func_t * rl_filename_dequoting_function` [Variable]
A pointer to a function that will remove application-specific quoting characters from a filename before completion is attempted, so those characters do not interfere with matching the text against names in the filesystem. It is called with *text*, the text of the word to be dequoted, and *quote_char*, which is the quoting character that delimits the filename (usually '`'' or '`"`'). If *quote_char* is zero, the filename was not in an embedded string.

`rl_linebuf_func_t * rl_char_is_quoted_p` [Variable]
A pointer to a function to call that determines whether or not a specific character in the line buffer is quoted, according to whatever quoting mechanism the program calling Readline uses. The function is called with two arguments: *text*, the text of the line, and *index*, the index of the character in the line. It is used to decide whether a character found in `rl_completer_word_break_characters` should be used to break words for the completer.

`rl_compignore_func_t * rl_ignore_some_completions_function` [Variable]
This function, if defined, is called by the completer when real filename completion is done, after all the matching names have been generated. It is passed a `NULL` terminated array of matches. The first element (`matches[0]`) is the maximal substring common to all matches. This function can re-arrange the list of matches as required, but each element deleted from the array must be freed.

`rl_icppfunc_t * rl_directory_completion_hook` [Variable]
This function, if defined, is allowed to modify the directory portion of filenames Readline completes. It could be used to expand symbolic links or shell variables in pathnames. It is called with the address of a string (the current directory name) as an argument, and may modify that string. If the string is replaced with a new string, the old value should be freed. Any modified directory name should have a trailing slash. The modified value will be used as part of the completion, replacing the directory portion of the pathname the user typed. At the least, even if no other expansion is performed, this function should remove any quote characters from the directory name, because its result will be passed directly to `opendir()`.

The directory completion hook returns an integer that should be non-zero if the function modifies its directory argument. The function should not modify the directory argument if it returns 0.

`rl_icppfunc_t * rl_directory_rewrite_hook;` [Variable]
If non-zero, this is the address of a function to call when completing a directory name. This function takes the address of the directory name to be modified as an argument. Unlike `rl_directory_completion_hook`, it only modifies the directory name used in `opendir`, not what is displayed when the possible completions are printed or inserted. It is called before rl_directory_completion_hook. At the least, even if no other expansion is performed, this function should remove any quote characters from the directory name, because its result will be passed directly to `opendir()`.

The directory rewrite hook returns an integer that should be non-zero if the function modfies its directory argument. The function should not modify the directory argument if it returns 0.

rl_icppfunc_t * rl_filename_stat_hook [Variable]

If non-zero, this is the address of a function for the completer to call before deciding which character to append to a completed name. This function modifies its filename name argument, and the modified value is passed to stat() to determine the file's type and characteristics. This function does not need to remove quote characters from the filename.

The stat hook returns an integer that should be non-zero if the function modfies its directory argument. The function should not modify the directory argument if it returns 0.

rl_dequote_func_t * rl_filename_rewrite_hook [Variable]

If non-zero, this is the address of a function called when reading directory entries from the filesystem for completion and comparing them to the partial word to be completed. The function should perform any necessary application or system-specific conversion on the filename, such as converting between character sets or converting from a filesystem format to a character input format. The function takes two arguments: *fname*, the filename to be converted, and *fnlen*, its length in bytes. It must either return its first argument (if no conversion takes place) or the converted filename in newly-allocated memory. The converted form is used to compare against the word to be completed, and, if it matches, is added to the list of matches. Readline will free the allocated string.

rl_compdisp_func_t * rl_completion_display_matches_hook [Variable]

If non-zero, then this is the address of a function to call when completing a word would normally display the list of possible matches. This function is called in lieu of Readline displaying the list. It takes three arguments: (char ***matches*, int *num_matches*, int *max_length*) where *matches* is the array of matching strings, *num_matches* is the number of strings in that array, and *max_length* is the length of the longest string in that array. Readline provides a convenience function, rl_display_match_list, that takes care of doing the display to Readline's output stream. You may call that function from this hook.

const char * rl_basic_word_break_characters [Variable]

The basic list of characters that signal a break between words for the completer routine. The default value of this variable is the characters which break words for completion in Bash: " \t\n\"\\'`@$><=;|&{(".

const char * rl_basic_quote_characters [Variable]

A list of quote characters which can cause a word break.

const char * rl_completer_word_break_characters [Variable]

The list of characters that signal a break between words for rl_complete_internal(). The default list is the value of rl_basic_word_break_characters.

`rl_cpvfunc_t * rl_completion_word_break_hook` [Variable]

If non-zero, this is the address of a function to call when Readline is deciding where to separate words for word completion. It should return a character string like `rl_completer_word_break_characters` to be used to perform the current completion. The function may choose to set `rl_completer_word_break_characters` itself. If the function returns `NULL`, `rl_completer_word_break_characters` is used.

`const char * rl_completer_quote_characters` [Variable]

A list of characters which can be used to quote a substring of the line. Completion occurs on the entire substring, and within the substring `rl_completer_word_break_characters` are treated as any other character, unless they also appear within this list.

`const char * rl_filename_quote_characters` [Variable]

A list of characters that cause a filename to be quoted by the completer when they appear in a completed filename. The default is the null string.

`const char * rl_special_prefixes` [Variable]

The list of characters that are word break characters, but should be left in *text* when it is passed to the completion function. Programs can use this to help determine what kind of completing to do. For instance, Bash sets this variable to `"$@"` so that it can complete shell variables and hostnames.

`int rl_completion_query_items` [Variable]

Up to this many items will be displayed in response to a possible-completions call. After that, readline asks the user if she is sure she wants to see them all. The default value is 100. A negative value indicates that Readline should never ask the user.

`int rl_completion_append_character` [Variable]

When a single completion alternative matches at the end of the command line, this character is appended to the inserted completion text. The default is a space character (' '). Setting this to the null character ('`\0`') prevents anything being appended automatically. This can be changed in application-specific completion functions to provide the "most sensible word separator character" according to an application-specific command line syntax specification. It is set to the default before any application-specific completion function is called, and may only be changed within such a function.

`int rl_completion_suppress_append` [Variable]

If non-zero, *rl_completion_append_character* is not appended to matches at the end of the command line, as described above. It is set to 0 before any application-specific completion function is called, and may only be changed within such a function.

`int rl_completion_quote_character` [Variable]

When Readline is completing quoted text, as delimited by one of the characters in *rl_completer_quote_characters*, it sets this variable to the quoting character found. This is set before any application-specific completion function is called.

`int rl_completion_suppress_quote` [Variable]

If non-zero, Readline does not append a matching quote character when performing completion on a quoted string. It is set to 0 before any application-specific completion function is called, and may only be changed within such a function.

`int rl_completion_found_quote` [Variable]

When Readline is completing quoted text, it sets this variable to a non-zero value if the word being completed contains or is delimited by any quoting characters, including backslashes. This is set before any application-specific completion function is called.

`int rl_completion_mark_symlink_dirs` [Variable]

If non-zero, a slash will be appended to completed filenames that are symbolic links to directory names, subject to the value of the user-settable *mark-directories* variable. This variable exists so that application-specific completion functions can override the user's global preference (set via the *mark-symlinked-directories* Readline variable) if appropriate. This variable is set to the user's preference before any application-specific completion function is called, so unless that function modifies the value, the user's preferences are honored.

`int rl_ignore_completion_duplicates` [Variable]

If non-zero, then duplicates in the matches are removed. The default is 1.

`int rl_filename_completion_desired` [Variable]

Non-zero means that the results of the matches are to be treated as filenames. This is *always* zero when completion is attempted, and can only be changed within an application-specific completion function. If it is set to a non-zero value by such a function, directory names have a slash appended and Readline attempts to quote completed filenames if they contain any characters in `rl_filename_quote_characters` and `rl_filename_quoting_desired` is set to a non-zero value.

`int rl_filename_quoting_desired` [Variable]

Non-zero means that the results of the matches are to be quoted using double quotes (or an application-specific quoting mechanism) if the completed filename contains any characters in `rl_filename_quote_chars`. This is *always* non-zero when completion is attempted, and can only be changed within an application-specific completion function. The quoting is effected via a call to the function pointed to by `rl_filename_quoting_function`.

`int rl_attempted_completion_over` [Variable]

If an application-specific completion function assigned to `rl_attempted_completion_function` sets this variable to a non-zero value, Readline will not perform its default filename completion even if the application's completion function returns no matches. It should be set only by an application's completion function.

`int rl_sort_completion_matches` [Variable]

If an application sets this variable to 0, Readline will not sort the list of completions (which implies that it cannot remove any duplicate completions). The default value is 1, which means that Readline will sort the completions and, depending on the value of `rl_ignore_completion_duplicates`, will attempt to remove duplicate matches.

`int rl_completion_type` [Variable]

Set to a character describing the type of completion Readline is currently attempting; see the description of `rl_complete_internal()` (see Section 2.6.2 [Completion Functions], page 51) for the list of characters. This is set to the appropriate value before any application-specific completion function is called, allowing such functions to present the same interface as `rl_complete()`.

`int rl_completion_invoking_key` [Variable]

Set to the final character in the key sequence that invoked one of the completion functions that call `rl_complete_internal()`. This is set to the appropriate value before any application-specific completion function is called.

`int rl_inhibit_completion` [Variable]

If this variable is non-zero, completion is inhibited. The completion character will be inserted as any other bound to `self-insert`.

2.6.4 A Short Completion Example

Here is a small application demonstrating the use of the GNU Readline library. It is called `fileman`, and the source code resides in `examples/fileman.c`. This sample application provides completion of command names, line editing features, and access to the history list.

```
/* fileman.c -- A tiny application which demonstrates how to use the
   GNU Readline library.  This application interactively allows users
   to manipulate files and their modes. */

#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif

#include <sys/types.h>
#ifdef HAVE_SYS_FILE_H
#  include <sys/file.h>
#endif
#include <sys/stat.h>

#ifdef HAVE_UNISTD_H
#  include <unistd.h>
#endif

#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

#if defined (HAVE_STRING_H)
#  include <string.h>
#else /* !HAVE_STRING_H */
#  include <strings.h>
#endif /* !HAVE_STRING_H */

#ifdef HAVE_STDLIB_H
#  include <stdlib.h>
#endif

#include <time.h>

#include <readline/readline.h>
#include <readline/history.h>

extern char *xmalloc PARAMS((size_t));

/* The names of functions that actually do the manipulation. */
int com_list PARAMS((char *));
int com_view PARAMS((char *));
int com_rename PARAMS((char *));
int com_stat PARAMS((char *));
int com_pwd PARAMS((char *));
int com_delete PARAMS((char *));
int com_help PARAMS((char *));
int com_cd PARAMS((char *));
int com_quit PARAMS((char *));

/* A structure which contains information on the commands this program
   can understand. */

typedef struct {
  char *name; /* User printable name of the function. */
  rl_icpfunc_t *func; /* Function to call to do the job. */
  char *doc; /* Documentation for this function.  */
} COMMAND;
```

```
COMMAND commands[] = {
  { "cd", com_cd, "Change to directory DIR" },
  { "delete", com_delete, "Delete FILE" },
  { "help", com_help, "Display this text" },
  { "?", com_help, "Synonym for `help'" },
  { "list", com_list, "List files in DIR" },
  { "ls", com_list, "Synonym for `list'" },
  { "pwd", com_pwd, "Print the current working directory" },
  { "quit", com_quit, "Quit using Fileman" },
  { "rename", com_rename, "Rename FILE to NEWNAME" },
  { "stat", com_stat, "Print out statistics on FILE" },
  { "view", com_view, "View the contents of FILE" },
  { (char *)NULL, (rl_icpfunc_t *)NULL, (char *)NULL }
};

/* Forward declarations. */
char *stripwhite ();
COMMAND *find_command ();

/* The name of this program, as taken from argv[0]. */
char *progname;

/* When non-zero, this global means the user is done using this program. */
int done;

char *
dupstr (s)
     char *s;
{
  char *r;

  r = xmalloc (strlen (s) + 1);
  strcpy (r, s);
  return (r);
}

main (argc, argv)
     int argc;
     char **argv;
{
  char *line, *s;

  progname = argv[0];

  initialize_readline (); /* Bind our completer. */

  /* Loop reading and executing lines until the user quits. */
  for ( ; done == 0; )
    {
      line = readline ("FileMan: ");

      if (!line)
        break;

      /* Remove leading and trailing whitespace from the line.
         Then, if there is anything left, add it to the history list
         and execute it. */
```

```
      s = stripwhite (line);

      if (*s)
        {
          add_history (s);
          execute_line (s);
        }

      free (line);
    }
  exit (0);
}

/* Execute a command line. */
int
execute_line (line)
     char *line;
{
  register int i;
  COMMAND *command;
  char *word;

  /* Isolate the command word. */
  i = 0;
  while (line[i] && whitespace (line[i]))
    i++;
  word = line + i;

  while (line[i] && !whitespace (line[i]))
    i++;

  if (line[i])
    line[i++] = '\0';

  command = find_command (word);

  if (!command)
    {
      fprintf (stderr, "%s: No such command for FileMan.\n", word);
      return (-1);
    }

  /* Get argument to command, if any. */
  while (whitespace (line[i]))
    i++;

  word = line + i;

  /* Call the function. */
  return ((*(command->func)) (word));
}

/* Look up NAME as the name of a command, and return a pointer to that
   command.  Return a NULL pointer if NAME isn't a command name. */
COMMAND *
find_command (name)
     char *name;
{
```

```
  register int i;

  for (i = 0; commands[i].name; i++)
    if (strcmp (name, commands[i].name) == 0)
      return (&commands[i]);

  return ((COMMAND *)NULL);
}

/* Strip whitespace from the start and end of STRING.  Return a pointer
   into STRING. */
char *
stripwhite (string)
     char *string;
{
  register char *s, *t;

  for (s = string; whitespace (*s); s++)
    ;

  if (*s == 0)
    return (s);

  t = s + strlen (s) - 1;
  while (t > s && whitespace (*t))
    t--;
  *++t = '\0';

  return s;
}

/* **************************************************************** */
/*                                                                  */
/*                  Interface to Readline Completion                */
/*                                                                  */
/* **************************************************************** */

char *command_generator PARAMS((const char *, int));
char **fileman_completion PARAMS((const char *, int, int));

/* Tell the GNU Readline library how to complete.  We want to try to complete
   on command names if this is the first word in the line, or on filenames
   if not. */
initialize_readline ()
{
  /* Allow conditional parsing of the ~/.inputrc file. */
  rl_readline_name = "FileMan";

  /* Tell the completer that we want a crack first. */
  rl_attempted_completion_function = fileman_completion;
}

/* Attempt to complete on the contents of TEXT.  START and END bound the
   region of rl_line_buffer that contains the word to complete.  TEXT is
   the word to complete.  We can use the entire contents of rl_line_buffer
   in case we want to do some simple parsing.  Return the array of matches,
   or NULL if there aren't any. */
char **
```

```
fileman_completion (text, start, end)
     const char *text;
     int start, end;
{
  char **matches;

  matches = (char **)NULL;

  /* If this word is at the start of the line, then it is a command
     to complete.  Otherwise it is the name of a file in the current
     directory. */
  if (start == 0)
    matches = rl_completion_matches (text, command_generator);

  return (matches);
}

/* Generator function for command completion.  STATE lets us know whether
   to start from scratch; without any state (i.e. STATE == 0), then we
   start at the top of the list. */
char *
command_generator (text, state)
     const char *text;
     int state;
{
  static int list_index, len;
  char *name;

  /* If this is a new word to complete, initialize now.  This includes
     saving the length of TEXT for efficiency, and initializing the index
     variable to 0. */
  if (!state)
    {
      list_index = 0;
      len = strlen (text);
    }

  /* Return the next name which partially matches from the command list. */
  while (name = commands[list_index].name)
    {
      list_index++;

      if (strncmp (name, text, len) == 0)
        return (dupstr(name));
    }

  /* If no names matched, then return NULL. */
  return ((char *)NULL);
}

/* **************************************************************** */
/*                                                                  */
/*                        FileMan Commands                          */
/*                                                                  */
/* **************************************************************** */

/* String to pass to system ().  This is for the LIST, VIEW and RENAME
   commands. */
```

```
static char syscom[1024];

/* List the file(s) named in arg. */
com_list (arg)
     char *arg;
{
  if (!arg)
    arg = "";

  sprintf (syscom, "ls -FClg %s", arg);
  return (system (syscom));
}

com_view (arg)
     char *arg;
{
  if (!valid_argument ("view", arg))
    return 1;

#if defined (__MSDOS__)
  /* more.com doesn't grok slashes in pathnames */
  sprintf (syscom, "less %s", arg);
#else
  sprintf (syscom, "more %s", arg);
#endif
  return (system (syscom));
}

com_rename (arg)
     char *arg;
{
  too_dangerous ("rename");
  return (1);
}

com_stat (arg)
     char *arg;
{
  struct stat finfo;

  if (!valid_argument ("stat", arg))
    return (1);

  if (stat (arg, &finfo) == -1)
    {
      perror (arg);
      return (1);
    }

  printf ("Statistics for `%s':\n", arg);

  printf ("%s has %d link%s, and is %d byte%s in length.\n",
  arg,
          finfo.st_nlink,
          (finfo.st_nlink == 1) ? "" : "s",
          finfo.st_size,
          (finfo.st_size == 1) ? "" : "s");
  printf ("Inode Last Change at: %s", ctime (&finfo.st_ctime));
```

```
  printf ("      Last access at: %s", ctime (&finfo.st_atime));
  printf ("    Last modified at: %s", ctime (&finfo.st_mtime));
  return (0);
}

com_delete (arg)
     char *arg;
{
  too_dangerous ("delete");
  return (1);
}

/* Print out help for ARG, or for all of the commands if ARG is
   not present. */
com_help (arg)
     char *arg;
{
  register int i;
  int printed = 0;

  for (i = 0; commands[i].name; i++)
    {
      if (!*arg || (strcmp (arg, commands[i].name) == 0))
        {
          printf ("%s\t\t%s.\n", commands[i].name, commands[i].doc);
          printed++;
        }
    }

  if (!printed)
    {
      printf ("No commands match `%s'.  Possibilties are:\n", arg);

      for (i = 0; commands[i].name; i++)
        {
          /* Print in six columns. */
          if (printed == 6)
            {
              printed = 0;
              printf ("\n");
            }

          printf ("%s\t", commands[i].name);
          printed++;
        }

      if (printed)
        printf ("\n");
    }
  return (0);
}

/* Change to the directory ARG. */
com_cd (arg)
     char *arg;
{
  if (chdir (arg) == -1)
    {
```

```
      perror (arg);
      return 1;
    }

  com_pwd ("");
  return (0);
}

/* Print out the current working directory. */
com_pwd (ignore)
     char *ignore;
{
  char dir[1024], *s;

  s = getcwd (dir, sizeof(dir) - 1);
  if (s == 0)
    {
      printf ("Error getting pwd: %s\n", dir);
      return 1;
    }

  printf ("Current directory is %s\n", dir);
  return 0;
}

/* The user wishes to quit using this program.  Just set DONE non-zero. */
com_quit (arg)
     char *arg;
{
  done = 1;
  return (0);
}

/* Function which tells you that you can't do this. */
too_dangerous (caller)
     char *caller;
{
  fprintf (stderr,
           "%s: Too dangerous for me to distribute.  Write it yourself.\n",
           caller);
}

/* Return non-zero if ARG is a valid argument for CALLER, else print
   an error message and return zero. */
int
valid_argument (caller, arg)
     char *caller, *arg;
{
  if (!arg || !*arg)
    {
      fprintf (stderr, "%s: Argument required.\n", caller);
      return (0);
    }

  return (1);
}
```

Appendix A GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

Function and Variable Index

K

M

N

O

P

Q

R

S

T

U

V

Y