GNU Readline Library

Edition 4.0, for `Readline Library` Version 4.0.
December 1998

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

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

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

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. If you do not have a meta key, the identical keystroke can be generated by typing ⟨ESC⟩ *first*, and then typing ⟨k⟩. Either process is known as *metafying* the ⟨k⟩ key.

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

In addition, several keys have their own names. Specifically, ⟨DEL⟩, ⟨ESC⟩, ⟨LFD⟩, ⟨SPC⟩, ⟨RET⟩, and ⟨TAB⟩ all stand for themselves when seen in this text, or in an init file (see Section 1.3 [Readline Init File], page 3).

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

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

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

⟨C-b⟩	Move back one character.

⟨C-f⟩	Move forward one character.
⟨DEL⟩	Delete the character to the left of the cursor.
⟨C-d⟩	Delete the character underneath the cursor.
Printing characters	Insert the character into the line at the cursor.
⟨C-_⟩	Undo the last editing command. You can undo all the way back to an empty line.

1.2.2 Readline Movement Commands

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

⟨C-a⟩	Move to the start of the line.
⟨C-e⟩	Move to the end of the line.
⟨M-f⟩	Move forward a word, 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. If the description for a command says that it 'kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

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

Here is the list of commands for killing text.

⟨C-k⟩	Kill the text from the current cursor position to the end of the line.
⟨M-d⟩	Kill from the cursor to the end of the current word, or if between words, to the end of the next word.
⟨M-DEL⟩	Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.
⟨C-w⟩	Kill from the cursor to the previous whitespace. This is different than ⟨M-DEL⟩ because the word boundaries differ.

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`'.

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. 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-s⟩ or ⟨C-r⟩ 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.

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 in his home directory. The name of this file is taken from the value of the environment variable `INPUTRC`. If that variable is unset, the default is '`~/.inputrc`'.

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

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

1.3.1 Readline Init File Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a '`#`' are comments. Lines beginning with a '`$`' indicate conditional constructs (see Section 1.3.2 [Conditional Init Constructs], page 8). Other lines denote variable settings and key bindings.

Variable Settings

You can modify the run-time behavior of Readline by altering the values of variables in Readline using the `set` command within the init file. Here is how to change from the default Emacs-like key binding to use `vi` line editing commands:

```
set editing-mode vi
```

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

`bell-style`

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

`comment-begin`

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

`completion-ignore-case`

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

`completion-query-items`

The number of possible completions that determines when the user is asked whether he wants to see the list of possibilities. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. 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 prepending an ⟨ESC⟩ character, converting them to a meta-prefixed key sequence. The default value is '`on`'.

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

`editing-mode`
The `editing-mode` variable controls which default set of key bindings is used. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can be set to either '`emacs`' or '`vi`'.

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

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

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

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

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

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

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

`mark-modified-lines`

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

`output-meta`

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

`print-completions-horizontally`

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

`show-all-if-ambiguous`

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

`visible-stats`

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

Key Bindings

The syntax for controlling key bindings in the init file is simple. First you have to know 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 the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the init file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: *function-name* or *macro*

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

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

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

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

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

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

In the above example, ⟨C-u⟩ is bound to the function `universal-argument` (just as it was in the first example), ‘⟨C-x⟩ ⟨C-r⟩’ is bound to the function `re-read-init-file`, and ‘⟨ESC⟩ ⟨[⟩ ⟨1⟩ ⟨1⟩ ⟨~⟩’ is bound to insert the text ‘`Function Key 1`’.

The following GNU Emacs style escape sequences are available when specifying key sequences:

`\C-`	control prefix
`\M-`	meta prefix
`\e`	an escape character
`\\`	backslash
`\"`	⟨"⟩
`\'`	⟨'⟩

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

`\a`	alert (bell)
`\b`	backspace
`\d`	delete
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\`*nnn*	the character whose ASCII code is the octal value *nnn* (one to three digits)
`\x`*nnn*	the character whose ASCII code is the hexadecimal value *nnn* (one to three digits)

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

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

1.3.2 Conditional Init Constructs

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

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

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

- `term` The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the '`=`' is tested against both the full name of the terminal and the portion of the terminal name before the first '`-`'. This allows `sun` to match both `sun` and `sun-cmd`, for instance.

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

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

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

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

1.3.3 Sample Init File

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

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

#
# Set various bindings for emacs mode.

set editing-mode emacs

$if mode=emacs

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

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

```
#"\M-\C-[B":      next-history

C-q: quoted-insert

$endif

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

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

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

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

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

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

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

# For FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
```

```
"\M-.": yank-last-arg
$endif
```

1.4 Bindable Readline Commands

This section describes Readline commands that may be bound to key sequences.

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 this, or the previous, word. Words are composed of letters and digits.

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

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

1.4.2 Commands For Manipulating The History

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

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

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

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

`end-of-history (M->)`
Move to the end of the input history, i.e., the line currently being entered.

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

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

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

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

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

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

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

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

1.4.3 Commands For Changing Text

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

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

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

`quoted-insert (C-q, C-v)`
Add the next character typed to the line verbatim. This is how to insert key sequences like ⟨C-q⟩, for example.

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

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

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

`transpose-words (M-t)`
Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

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

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

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

1.4.4 Killing And Yanking

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

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

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

`kill-whole-line ()`

Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word (M-d)`

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

`backward-kill-word (M-DEL)`

Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`

Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`

Delete all spaces and tabs around point. By default, this is unbound.

`kill-region ()`

Kill the text between the point and the *mark* (saved cursor position). This text is referred to as the *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 the current cursor position.

`yank-pop (M-y)`

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

1.4.5 Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`

Add this digit to the argument already accumulating, or start a new argument. ⟨M-⟩ starts a negative argument.

`universal-argument ()`

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

1.4.6 Letting Readline Type For You

`complete (TAB)`

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

`possible-completions (M-?)`

List the possible completions of the text before the cursor.

`insert-completions (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 and the original text is restored. An argument of *n* moves *n* positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to `TAB`, but is unbound by default.

`delete-char-or-list ()`

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

1.4.7 Keyboard Macros

`start-kbd-macro (C-x ()`

Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro (C-x ))`

Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro (C-x e)`

Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

1.4.8 Some Miscellaneous Commands

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

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

`do-uppercase-version (M-a, M-b, M-`*x*`, ...)`
: If the metafied character *x* is lowercase, run the command that is bound to the corresponding uppercase character.

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

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

`revert-line (M-r)`
: Undo all changes made to this line. This is like 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 current point. If a numeric argument is supplied, the mark is set to that position.

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

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

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

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

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

`dump-variables ()`
Print all of the settable variables and their values to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

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

1.5 Readline vi Mode

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

In order to switch interactively between `emacs` and `vi` editing modes, use the command M-C-j (toggle-editing-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 and then reads and returns a single line of text from the user. The line `readline` returns is allocated with `malloc ()`; you should `free ()` the line when you are done with it. The declaration for `readline` in ANSI C is

`char *readline (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.

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*`, int (*`*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 32).

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.

2.2.1 The Function Type

For readabilty, we declare a new type of object, called *Function*. A `Function` is a C function which returns an `int`. The type declaration for `Function` is:

`typedef int Function ();`

The reason for declaring this new type is to make it easier to write code describing pointers to C functions. Let us say we had a variable called *func* which was a pointer to a function. Instead of the classic C declaration

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

we may write

```
Function *func;
```

Similarly, there are

`typedef void VFunction ();`
`typedef char *CPFunction ();` and
`typedef char **CPPFunction ();`

for functions returning no value, `pointer to char`, and `pointer to pointer to char`, respectively.

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

```
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.

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 26.

`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_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_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.

`char *` **rl_library_version** Variable

The version number of this revision of the library.

`char *` **rl_terminal_name** Variable

The terminal type, used for initialization.

`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 8).

`FILE *` **rl_instream** Variable

The stdio stream from which Readline reads input.

`FILE *` **rl_outstream** Variable

The stdio stream to which Readline performs output.

`Function *` **rl_startup_hook** Variable

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

`Function *` **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.

`Function *` **rl_event_hook** Variable

If non-zero, this is the address of a function to call periodically when readline is waiting for terminal input.

`Function *` **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 [Utility Functions], page 28).

`VFunction *` **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 27).

`Keymap` **rl_executing_keymap** Variable

This variable is set to the keymap (see Section 2.4.2 [Keymaps], page 23) 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 23) in which the last key binding occurred.

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** `(char *name, Function *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** `()` Function

Returns a new, empty keymap. The space for the keymap is allocated with `malloc ()`; you should `free ()` it when you are done.

`Keymap` **rl_copy_keymap** `(Keymap map)` Function

Return a new keymap which is a copy of *map*.

`Keymap` **rl_make_keymap** `()` 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 *keymap*.

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

`Keymap` **rl_get_keymap** `()` 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** `(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 3).

`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 3).

2.4.3 Binding Keys

You associate keys 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.

These functions manage key bindings.

`int` **rl_bind_key** `(int key, Function *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, Function *function, Keymap map)` Function

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

`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** `(Function *function, Keymap map)` Function

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

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

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

`int` **rl_generic_bind** (`int type, 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 3).

`int` **rl_read_init_file** (`char *filename`) Function

Read keybindings and variable assignments from *filename* (see Section 1.3 [Readline Init File], page 3).

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.

`Function *` **rl_named_function** (`char *name`) Function

Return the function with name *name*.

`Function *` **rl_function_of_keyseq** (`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 it (one of `ISFUNC`, `ISKMAP`, or `ISMACR`).

`char **` **rl_invoking_keyseqs** (`Function *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** (`Function *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** () Function

Print the names of all bindable Readline functions to `rl_outstream`.

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. I could use an undo function for the stock market.

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 undo 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** () 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** () 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` **free_undo_list** () Function

Free the existing undo list.

`int` **rl_do_undo** () 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** () Function

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

`int` **rl_forced_update_display** () Function

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

`int` **rl_on_new_line** () Function

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

`int` **rl_reset_line_state** () Function

Reset the display state to a clean state and redisplay the current line starting on a new line.

`int` **rl_message** (`va_alist`) Function

The arguments are a string as would be supplied to `printf`. The resulting string is displayed in the *echo area*. The echo area is also used to display numeric arguments and search strings.

`int` **rl_clear_message** () Function

Clear the message in the echo area.

`void` **rl_save_prompt** () 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** () Function

Restore the local Readline prompt display state saved by the most recent call to `rl_save_prompt`.

2.4.7 Modifying Text

`int` **rl_insert_text** (`char *text`) Function

Insert *text* into the line at the current cursor position.

`int` **rl_delete_text** (`int start, int end`) Function

Delete the text between *start* and *end* in the current line.

`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.

2.4.8 Utility Functions

`int` **rl_read_key** `()` Function

Return the next character available. This handles input inserted into the input stream via *pending input* (see Section 2.3 [Readline Variables], page 21) and `rl_stuff_char ()`, macros, and characters read from the keyboard.

`int` **rl_getc** `(FILE *)` Function

Return the next character available from 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 ()`.

`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** `()` Function

Initialize or re-initialize Readline's internal state.

`int` **rl_reset_terminal** `(char *terminal_name)` Function

Reinitialize Readline's idea of the terminal settings using *terminal_name* as the terminal type (e.g., `vt100`).

`int` **alphabetic** `(int c)` Function

Return 1 if *c* is an alphabetic character.

`int` **numeric** `(int c)` Function

Return 1 if *c* is a numeric character.

`int` **ding** `()` Function

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

`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).

The following are implemented as macros, defined in `chartypes.h`.

`int` **uppercase_p** `(int c)` Function

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

`int` **lowercase_p** `(int c)` Function

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

`int` **digit_p** (`int c`) Function

Return 1 if *c* is a numeric character.

`int` **to_upper** (`int c`) Function

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

`int` **to_lower** (`int c`) Function

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

`int` **digit_value** (`int c`) Function

If *c* is a number, return the value it represents.

2.4.9 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 accomodate 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** (`char *prompt, Vfunction *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 callback when a complete line of input has been entered.

`void` **rl_callback_read_char** () 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 saved by `rl_callback_handler_install` to process the line. `EOF` is indicated by calling *lhandler* with a `NULL` line.

`void` **rl_callback_handler_remove** () Function

Restore the terminal to its initial state and remove the line handler. This may be called from within a callback as well as independently.

2.4.10 An 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 (uppercase_p (rl_line_buffer[i]))
        rl_line_buffer[i] = to_lower (rl_line_buffer[i]);
      else if (lowercase_p (rl_line_buffer[i]))
        rl_line_buffer[i] = 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);
  }
```

2.5 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 a signal is received 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`, `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 ()`).

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 state, 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.

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`, `SIGALRM`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU`.

The default value of `rl_catch_signals` is 1.

`int` **rl_catch_sigwinch** Variable

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

The default value of `rl_catch_sigwinch` 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.

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

`void` **rl_resize_terminal** (`void`) Function

Update Readline's internal screen size.

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`, `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 Readline functions intended for interactive use: *count* and *invoking_key*. It isolates the word to be completed and calls `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 `completion_matches ()` uses your *generator* function to generate the list of possible matches, and then returns the array of these matches. You should place the address of your generator function in `rl_completion_entry_function`.
3. The generator function is called repeatedly from `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. When the generator function returns `(char *)NULL` this signals `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.

`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 `completion_matches ()`). The default is to do filename completion.

`Function *` **rl_completion_entry_function** Variable

This is a pointer to the generator function for `completion_matches ()`. If the value of `rl_completion_entry_function` is `(Function *)NULL` then the default filename generator function, `filename_completion_function ()`, is used.

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.

`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 `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 '`*`'.

`char **` **completion_matches** `(char *text, CPFunction *entry_func)` Function

Returns an array of `(char *)` which is a list of completions for *text*. If there are no completions, returns `(char **)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 *` **filename_completion_function** `(char *text, int state)` Function

A generator function for filename completion in the general case. Note that completion in Bash is a little different because of all the pathnames that must be followed when looking up completions for a command. The Bash source is a useful reference for writing custom completion functions.

`char *` **username_completion_function** `(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

`Function *` **rl_completion_entry_function** Variable

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

`CPPFunction *` **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` saying what the boundaries of *text* are. 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.

`CPFunction *` **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.

`CPFunction *` **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.

`Function *` **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.

`int` **rl_completion_query_items** Variable

Up to this many items will be displayed in response to a possible-completions call. After that, we ask the user if she is sure she wants to see them all. The default value is 100.

`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, i.e., `" \t\n\"\\’‘@$><=;|&{("`.

`char *` **rl_basic_quote_characters** Variable

List of quote characters which can cause a word break.

`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`.

`char *` **rl_completer_quote_characters** Variable

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.

`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.

`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_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 custom completion functions to provide the "most sensible word separator character" according to an application-specific command line syntax specification.

`int` **rl_ignore_completion_duplicates** Variable

If non-zero, then disallow duplicates in the matches. 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 on entry, and can only be changed within a completion entry generator function. If it is set to a non-zero value, directory names have a slash appended and Readline attempts to quote completed filenames if they contain any embedded word break characters.

`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 on entry, and can only be changed within a completion entry generator function. The quoting is effected via a call to the function pointed to by `rl_filename_quoting_function`.

`int` **rl_inhibit_completion** Variable

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

`Function *` **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.

`Function *` **rl_directory_completion_hook** Variable

This function, if defined, is allowed to modify the directory portion of filenames Readline completes. It is called with the address of a string (the current directory name) as an argument. It could be used to expand symbolic links or shell variables in pathnames.

`VFunction *` **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. That function may be called from this hook.

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. */

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/errno.h>

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

extern char *getwd ();
extern char *xmalloc ();

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

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

typedef struct {
  char *name; /* User printable name of the function. */
  Function *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, (Function *)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)
     int 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 ();
char **fileman_completion ();

/* 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 = (CPPFunction *)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)
     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 = 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)
     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;

  sprintf (syscom, "more %s", arg);
  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 = getwd (dir);
  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);
}
```

Concept Index

Function and Variable Index

Table of Contents