GNU Readline Library

Brian Fox
Free Software Foundation
Version 1.1
April 1991

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

Appendix A Command Line Editing

This text describes GNU's command line editing interface.

A.1 Introduction to Line Editing

The following paragraphs describe the notation we use to represent keystrokes.

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

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

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

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

A.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.

A.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 ⟨DEL⟩ 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. Aftwerwards, 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 get '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 get '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 itself into the line at the cursor.

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

A.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.

⟨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.

A.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* 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.

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 to the start of the previous word, or if between words, to the start of the previous word.

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

And, here is how to *yank* the text back into the line.

⟨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⟩.

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 in one clean sweep. 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.

A.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' you type is a minus sign (⟨-⟩), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the ⟨C-d⟩ command an argument of 10, you could type ⟨M-1 0 C-d⟩.

A.3 Readline Init File

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

When a program which uses the Readline library starts up, the '`~/.inputrc`' file is read, and the keybindings 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.

A.3.1 Readline Init Syntax

There are only four constructs allowed in the '`~/.inputrc`' file:

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

```
set editing-mode vi
```

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

`editing-mode`
: The `editing-mode` variable controls which editing mode you are using. By default, GNU Readline starts up in Emacs editing mode,

where the keystrokes are most similar to Emacs. This variable can either be set to `emacs` or `vi`.

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

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

`prefer-visible-bell`
: If this variable is set to `On` it means to use a visible bell if one is available, rather than simply ringing the terminal bell. By default, the value is `Off`.

Key Bindings

The syntax for controlling keybindings in the '`~/.inputrc`' file is simple. First you have to know the *name* of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the '`~/.inputrc`' 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. Simply place the key sequence in double quotes. GNU Emacs style key escapes can be used, as in the following example:

```
"\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`'.

A.3.1.1 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.

`backward-word (⟨M-b⟩)`
: Move back to the start of this, or the previous, word.

`clear-screen (⟨C-l⟩)`
: Clear the screen leaving the current line at the top of the screen.

A.3.1.2 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 you are entering.

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

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

A.3.1.3 Commands For Changing Text

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

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

`quoted-insert` (C-q, C-v)
: Add the next character that you type to the line verbatim. This is how to insert things 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 point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. 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 all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

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

`capitalize-word` (M-c)
: Uppercase the first letter in the current (or following) word. With a negative argument, do the previous word, but do not move point.

A.3.1.4 Killing And Yanking

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

`backward-kill-line` ()
: Kill backward to the beginning of the line. This is normally 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.

`backward-kill-word` (⟨M-DEL⟩)
: Kill the word behind the cursor.

`unix-line-discard` (⟨C-u⟩)
: Kill the whole line the way ⟨C-u⟩ used to in Unix line input. The killed text is saved on the kill-ring.

`unix-word-rubout` (⟨C-w⟩)
: Kill the word the way ⟨C-w⟩ used to in Unix line input. The killed text is saved on the kill-ring. This is different than backward-kill-word because the word boundaries differ.

`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.

A.3.1.5 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` ()
: Do what ⟨C-u⟩ does in GNU Emacs. By default, this is not bound.

A.3.1.6 Letting Readline Type For You

`complete` (TAB)
: Attempt to do completion on the text before point. This is implementation defined. 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.

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

A.3.1.7 Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`
Read in the contents of your '`~/.inputrc`' file, and incorporate any bindings found there.

`abort (C-g)`
Stop running the current editing command.

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

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

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

A.3.2 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.

In order to switch interactively between GNU Emacs and `vi` editing modes, use the command M-C-j (toggle-editing-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 'edit' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with 'k', and following lines with 'j', and so forth.

1 Programming with GNU Readline

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

1.1 Default Behaviour

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 ()`.

The function `readline` prints a prompt and then reads and returns a single line of text from the user. The line which `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 which is returned has the final newline removed, so only the text of the line 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 was 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 polite to avoid saving empty lines on the history list, since it is rare than someone has a burning need to reuse a blank line. Here is a function which usefully replaces the standard `gets ()` library function:

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

The above code 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 2 arguments; *key* is the character that you want to bind, and *function* is the address of the function to run when *key* is pressed. Binding ⟨TAB⟩ to `rl_insert ()` makes ⟨TAB⟩ just insert itself.

`rl_bind_key ()` returns non-zero if *key* is not a valid ASCII character code (between 0 and 255).

```
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, etc.

1.2 Custom Functions

Readline provides a great 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 in within the Readline library which allow a user program to add customized functionality to Readline.

1.2.1 The Function Type

For the sake of readabilty, we declare a new type of object, called *Function*. A `Function` is a C language 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 have

```
Function *func;
```

1.2.2 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:

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 already. If you need to do more or different things than adding a function to Readline, you may need to use the underlying functions described below.

1.2.3 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.

1.2.4 Binding Keys

You associate keys with functions through the keymap. Here are functions for doing that.

int rl_bind_key (*int key, Function *function*) Function

Binds *key* to *function* in the currently selected 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

Make *key* do nothing in the currently selected keymap. Returns non-zero in case of error.

int rl_unbind_key_in_map (*int key, Keymap map*) Function

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

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*; right now this can be a function (`ISFUNC`), a macro (`ISMACR`), or a keymap (`ISKMAP`). This makes new keymaps as necessary. The initial place to do bindings is in *map*.

1.2.5 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 gathered so far.

char *rl_line_buffer Variable

This is the line gathered so far. You are welcome to modify the contents of this, but see Undoing, below.

int rl_point Variable

The offset of the current cursor position in *rl_line_buffer*.

int rl_end Variable

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

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, other functions as a flag, and 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 a negative argument as well as a positive argument. At the very least, it should be aware that it can be passed a negative argument.

1.2.6 Allowing Undoing

Supporting the undo command is a painless thing to do, and makes your functions much more useful to the end user. 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 it calls `rl_insert_text ()` or `rl_delete_text ()` to do it, then undoing is already done for you automatically, and you can safely skip this section.

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

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 they could be direct calls to `rl_add_undo ()`.

rl_end_undo_group () Function

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

Finally, if you neither insert nor delete text, but directly modify the existing text (e.g. change its case), you 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.

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 subsequent to this call you will modify that range of text in some way.

1.2.7 An Example

Here is a function which changes lowercase characters to the uppercase equivalents, and uppercase characters to the lowercase equivalents. If this function was bound to '`M-c`', then typing '`M-c`' would change the case of the character under point. Typing '`10 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. */
invert_case_line (count, key)
     int count, key;
{
  register int start, end;

  start = rl_point;

  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 = -1;

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

  if (start == end)
    return;

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

  for (; start != end; start += direction)
    {
      if (uppercase_p (rl_line_buffer[start]))
        rl_line_buffer[start] = to_lower (rl_line_buffer[start]);
      else if (lowercase_p (rl_line_buffer[start]))
        rl_line_buffer[start] = to_upper (rl_line_buffer[start]);
    }
  /* Move point to on top of the last character changed. */
  rl_point = end - direction;
}
```

1.3 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 either commands, or data, or both commands and data. The following sections describe how your program and Readline cooperate to provide this service to end users.

1.3.1 How Completing Works

In order to complete some text, the full list of possible completions must be available. That is to say, it is not possible to accurately expand a partial word without knowing what all of the possible words that make sense in that context are. The GNU Readline library provides the user interface to completion, and additionally, 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 those functions must do, and provides an example function.

There are three major functions used to perform completion:

1. The user-interface function `rl_complete ()`. This function is called interactively with the same calling conventions as other functions in readline intended for interactive use; i.e. *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 or actually performs the completion, depending on which behaviour 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, 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.

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.

Note that `rl_complete ()` has the identical calling conventions as any other key-invokable function; this is because by default it is bound to the '`TAB`' key.

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 is used, namely `filename_entry_function ()`.

1.3.2 Completion Functions

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

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.

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. This just calls `rl_complete_internal ()` with an argument of '`TAB`'.

rl_possible_completions () Function

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

char **completion_matches (*char *text, char *(*entry_function) ()*) 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_function 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. It 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 the Bash shell is a little different because of all the pathnames that must be followed when looking up the completion for a command.

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 '~').

1.3.3 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.

Function *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` then `rl_complete ()` will call the value of `rl_completion_entry_function` to generate matches, otherwise the array of strings returned will be used.

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 contents of this variable is what breaks words in the Bash shell, i.e. `" \t\n\"\\'`@$><=;|&{("`.

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 contents of `rl_basic_word_break_characters`.

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.

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.

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 `(char *)` known as *matches* in the code. The 1st element (`matches[0]`) is the maximal substring that is common to all matches. This function can re-arrange the list of matches as required, but each deleted element of the array must be `free()`'d.

1.3.4 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 '`readline/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 <readline/readline.h>
#include <readline/history.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/errno.h>

/* 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 }
};

/* 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 = 0;
```

```
main (argc, argv)
     int argc;
     char **argv;
{
  progname = argv[0];

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

  /* Loop reading and executing lines until the user quits. */
  while (!done)
    {
      char *line;

      line = readline ("FileMan: ");

      if (!line)
        {
          done = 1;             /* Encountered EOF at top level. */
        }
      else
        {
          /* Remove leading and trailing whitespace from the line.
             Then, if there is anything left, add it to the history list
             and execute it. */
          stripwhite (line);

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

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

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

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

  word = line;

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

  command = find_command (word);

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

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

  word = line + i;

  /* Call the function. */
  (*(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. */
stripwhite (string)
     char *string;
{
  register int i = 0;
```

```
  while (whitespace (string[i]))
    i++;

  if (i)
    strcpy (string, string + i);

  i = strlen (string) - 1;

  while (i > 0 && whitespace (string[i]))
    i--;

  string[++i] = '\0';
}
```

```
/* **************************************************************** */
/*                                                                  */
/*                  Interface to Readline 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 ()
{
  char **fileman_completion ();

  /* Allow conditional parsing of the ~/.inputrc file. */
  rl_readline_name = "FileMan";

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

/* Attempt to complete on the contents of TEXT.  START and END show the
   region of TEXT that contains the word to complete.  We can use the
   entire line 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;
  char *command_generator ();

  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 (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);
  system (syscom);
}

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

  sprintf (syscom, "cat %s | more", arg);
  system (syscom);
}

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

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

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

  if (stat (arg, &finfo) == -1)
    {
```

```
      perror (arg);
      return;
    }

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

  printf ("%s has %d link%s, and is %d bytes in length.\n", arg,
          finfo.st_nlink, (finfo.st_nlink == 1) ? "" : "s",  finfo.st_size);
  printf ("      Created on: %s", ctime (&finfo.st_ctime));
  printf ("  Last access at: %s", ctime (&finfo.st_atime));
  printf ("Last modified at: %s", ctime (&finfo.st_mtime));
}

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

/* 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");
    }
}

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

  com_pwd ("");
}

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

  (void) getwd (dir);

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

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

/* 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

U

Y

Table of Contents