Bash Features

Cursory Documentation for Bash

Brian Fox, Free Software Foundation

1 Bourne Shell Style Features

Bash is an acronym for Bourne Again SHell, the traditional Unix shell written by Stephen Bourne. All of the Bourne shell builtin commands are available in Bash, and the rules for evaluation and quoting are taken from the Posix 1003 specification for the 'standard' Unix shell.

The shell builtin control features are briefly discussed here.

1.1 Looping Constructs

Note that whereever you see an '`;`' in the description of a command's syntax, it can be replaced indiscriminently with newlines.

Bash supports the following looping constructs.

`until` The syntax of the `until` command is:

> `until` *test-commands*`; do` *consequent-commands*`; done`

Execute *consequent-commands* as long as the final command in *test-commands* has an exit status which is not zero.

`while` The syntax of the `while` command is:

> `while` *test-commands*`; do` *consequent-commands*`; done`

Execute *consequent-commands* as long as the final command in *test-commands* has an exit status of zero.

`for` The syntax of the for command is:

> `for` *name* `[in` *words* `...]; do` *commands*`; done`

Execute *commands* for each member in *words*, with *name* bound to the current member. If "`in` *words*" is not present, "`in "$@"`" is assumed.

1.2 Conditional Constructs

`if` The syntax of the `if` command is:

```
if test-commands; then
  consequent-commands;
[else alternate-consequents;]
```

```
fi
```

Execute *consequent-commands* only if the final command in *test-commands* has an exit status of zero. If "`else` *alternate-consequents*" is present, and the final command in *test-commands* has a non-zero exit status, then execute *alternate-consequents*.

`case` The syntax of the `case` command is:

`case` *word* `in` [*pattern* `[|` *pattern*`]...)` *commands* `;;]... esac`

Selectively execute *commands* based upon *word* matching *pattern*. The '`|`' is used to separate multiple patterns.

Here is an example using `case` in a script that could be used to describe an interesting feature of an animal:

```
echo -n "Enter the name of an animal:"
read ANIMAL
echo -n "The $ANIMAL has "
case $ANIMAL in
  horse | dog | cat) echo -n "four";;
  man | kangaroo ) echo -n "two";;
  *) echo -n "an unknown number of";;
esac
echo "legs."
```

2 (T)C-Shell Style Features

The C-Shell `csh` was created by Bill Joy at UC Berkeley. It is generally considered to have better features for interactive use than the Bourne shell. Some of the `csh` features present in Bash include job control, history expansion, 'protected' redirection, and several variables for controlling the interactive behaviour of the shell (e.g. `ignoreeof`).

For details on history expansion, see Chapter 5 [Using History Interactively], page 15.

Bash has tilde (`~`) expansion, similar, but not identical, to that of `csh`. The following table shows what unquoted words beginning with a tilde expand to.

`~` The current value of `$HOME`.

`~/foo` `$HOME`/foo

`~fred/foo`
The subdirectory `foo` of the home directory of the user named `fred`.

`~+/foo` `$PWD`/foo

`~-` `$OLDPWD`/foo

Here is a list of the commands and variables whose meanings were taken from `csh`.

`pushd`

`pushd [`*dir* `| +`*n*`]`

Save the current directory on a list and then CD to DIR. With no arguments, exchanges the top two directories.

`+`*n* Brings the *n*th directory to the top of the list by rotating.

dir Makes the current working directory be the top of the stack, and then cd's to DIR. You can see the saved directory list with the 'dirs' command.

`popd`

`popd [+`*n*`]`

Pops the directory stack, and cd's to the new top directory. The elements are numbered from 0 starting at the first directory listed with `dirs`; i.e. `popd` is equivalent to `popd 0`.

`dirs`

`dirs`

Display the list of currently remembered directories. Directories find their way onto the list with the `pushd` command; you can get back up through the list with the `popd` command.

`history`

`history [`*n*`] [ [-w -r] [`*filename*`]]`

Display the history list with line numbers. Lines listed with with a `*` have been modified. Argument of *n* says to list only the last *n* lines. Argument `-w` means write out the current history file. `-r` means to read it instead. If *filename* is given, then use that file, else if `$HISTFILE` has a value, use that, else use '`~/.bash_history`'.

`ignoreeof`

If this variable is set, it represents the number of consecutive `EOF`s Bash will read before exiting. By default, Bash will exit upon reading an `EOF` character.

3 Korn Shell Style Features

`fc`

`fc [-e` *ename*`] [-nlr] [`*first*`] [`*last*`]`
`fc -s [`*pat=rep*`] [`*command*`]`

Fix Command. In the first form, a range of commands from *first* to *last* is selected from the history list. *First* and/or *last* may be specified as a string (to locate the most recent command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If *last* is not specified it is set to *first*. If *first* is not specified it is set to the previous command for editing and -16 for listing. If the `-l` flag is given, the commands are listed on standard output. The `-n` flag suppresses the command numbers when listing. The `-r` flag reverses the order of the listing. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the following variable expansion is used: `${FCEDIT:-${EDITOR:-vi}}`. This says to use the value of the `FCEDIT` variable if set, or the value of the `EDITOR` variable if that is set, or `vi` if neither is set. When editing is complete, the edited commands are echoed and executed.

In the second form, *command* is re-executed after the substitution *old=new* is performed.

A useful alias to use with the `fc` command is `r='fc -s'`, so that typing `r cc` runs the last command beginning with `cc` and typing `r` re-executes the last command.

`typeset` The `typeset` command is supplied for compatability with the Korn shell, however, it has been made obsolete by the presence of the `declare` command, documented with the Bash specific features.

`type` Bash's `type` command is a superset of the `type` found in Korn shells; See Section 4.6 [Bash Builtins], page 10 for details.

4 Bash Specific Features

4.1 Shell Command Line Options

Along with the single character shell command-line options (See Section 4.2 [The Set Builtin], page 7) there are several other options that you can use. These options must appear on the command line before the single character command options to be recognized.

`-norc` Don't load `~`/.bashrc init file. (Default if shell name is 'sh').

`-rcfile` *filename*
Load *filename* init file (instead '`~/.bashrc`').

`-noprofile`
Don't load '`~/.bash_profile`' (nor '`/etc/profile`').

`-version` Display the version number of this shell.

`-login` Make this shell act as if it were directly invoked from login. This is equivalent to "exec - bash" but can be issued from another shell, such as csh. If you wanted to replace your current login shell with a bash login shell, you would say "exec bash -login".

`-nobraceexpansion`
Do not preform curly brace expansion (foo`{`a,b`}` -> fooa foob).

`-nolinediting`
Do not use the GNU Readline library to read interactive text lines.

4.2 The Set Builtin

This builtin is so overloaded that it deserves its own section. So here it is.

`set`

`set [-aefhknotuvxldH] [`*arg* `...]`

`-a` Mark variables which are modified or created for export.

`-e` Exit immediately if a command exits with a non-zero status.

`-f` Disable file name generation (globbing).

`-k` All keyword arguments are placed in the environment for a command, not just those that precede the command name.

`-m` Job control is enabled.

`-n` Read commands but do not execute them.

`-o` *option-name*

Set the variable corresponding to *option-name*:

`allexport`
same as -a.

`braceexpand`
the shell will perform brace expansion.

`emacs` use an emacs-style line editing interface.

`errexit` same as -e.

`histexpand`
same as -H.

`ignoreeof`
the shell will not exit upon reading EOF.

`monitor` same as -m.

`noclobber`
disallow redirection to existing files.

`noexec` same as -n.

`noglob` same as -f.

`nohash` same as -d.

`notify` notify of job termination immediately.

`nounset` same as -u.

`verbose` same as -v.

`vi` use a vi-style line editing interface.

`xtrace` same as -x.

`-t` Exit after reading and executing one command.

`-u` Treat unset variables as an error when substituting.

`-v` Print shell input lines as they are read.

`-x` Print commands and their arguments as they are executed.

`-l` Save and restore the binding of the *name* in a `for` command.

`-d` Disable the hashing of commands that are looked up for execution. Normally, commands are remembered in a hash table, and once found, do not have to be looked up again.

`-H` Enable ! style history substitution. This flag is on by default.

Using '`+`' rather than '`-`' causes these flags to be turned off. The flags can also be used upon invocation of the shell. The current set of flags may be found in `$-`. The remaining *arg*s are positional parameters and are assigned, in order, to `$1`, `$2`, .. `$9`. If no *arg*s are given, all shell variables are printed.

4.3 Is This Shell Interactive?

You may wish to determine within a startup script whether Bash is running interactively or not. To do this, you examine the variable `$PS1`; it is unset in non-interactive shells, and set in interactive shells. Thus:

```
if [ "$PS1" = "" ]; then
  echo "This shell is not interactive"
else
  echo "This shell is interactive"
fi
```

You can ask an interactive Bash to not run your '`~/.bashrc`' file with the `-norc` flag. You can change the name of the '`~/.bashrc`' file to any other file name with `-rcfile` *filename*. You can ask Bash to not run your '`~/.bash_profile`' file with the `-noprofile` flag.

4.4 Controlling the Prompt

The value of the variable `$PROMPT_COMMAND` is examined just before Bash prints each toplevel prompt. If it is set and non-null, then the value is executed just as if you had typed it on the command line.

In addition, the following table describes the special characters which can appear in the `PS1` variable:

`\t`	the time.
`\d`	the date.
`\n`	CRLF.
`\s`	the name of the shell.
`\w`	the current working directory.
`\W`	the last element of PWD.
`\u`	your username.
`\h`	the hostname.
`\#`	the command number of this command.
`\!`	the history number of this command.
`\<octal>`	the character code in octal.
`\\`	a backslash.

4.5 Bash Startup Files

When and how Bash executes '`~/.bash_profile`', '`~/.bashrc`', and '`~/.bash_logout`'.

```
For Login shells:

    On logging in:
       If '/etc/profile' exists, then source it.

       If '~/.bash_profile' exists, then source it,
          else if '~/.bash_login' exists, then source it,
             else if '~/.profile' exists, then source it.

    On logging out:
       If '~/.bash_logout' exists, source it.

For non-login interactive shells:
    On starting up:
       If '~/.bashrc' exists, then source it.

For non-interactive shells:
    On starting up:
       If the environment variable ENV is non-null, source the
       file mentioned there.
```

So, typically, your `~/.bash_profile` contains the line

```
if [ -f '~/.bashrc' ]; then source '~/.bashrc'; fi
```

after (or before) any login specific initializations.

4.6 Bash Builtin Commands

`builtin`

`builtin` [*shell-builtin* [*args*]]

Run a shell builtin. This is useful when you wish to rename a shell builtin to be a function, but need the functionality of the builtin within the function itself.

`declare`

`declare [-frxi]` *name*`[=`*value*`]`

Declare variables and/or give them attributes. If no *name*s are given, then display the values of variables instead. `-f` means to use function names only. `-r` says to make

*name*s readonly. `-x` says to make *name*s export. `-i` says that the variable is to be treated as an integer; arithmetic evaluation (see `let`) will be done when the variable is assigned to. Using `+` instead of `-` turns off the attribute instead. When used in a function, makes *name*s local, as with the `local` command.

`type`

`type [-all] [-type | -path] [`*name* `...]`

For each *name*, indicate how it would be interpreted if used as a command name.

If the `-type` flag is used, `type` returns a single word which is one of "alias", "function", "builtin", "file" or "", if *name* is an alias, shell function, shell builtin, disk file, or unfound, respectively.

If the `-path` flag is used, `type` either returns the name of the disk file that would be exec'ed, or nothing if `-type` wouldn't return "file".

If the `-all` flag is used, returns all of the places that contain an executable named *file*. This includes aliases and functions, if and only if the `-path` flag is not also used.

`enable`

`enable [-n] [`*name* `...]`

Enable and disable builtin shell commands. This allows you to use a disk command which has the same name as a shell builtin. If `-n` is used, the *name*s become disabled. Otherwise *name*s are enabled. For example, to use the `test` found on your path instead of the shell builtin version, you type `enable -n test`.

`help`

`help [`*pattern*`]`

Display helpful information about builtin commands. If *pattern* is specified, gives detailed help on all commands matching *pattern*, otherwise a list of the builtins is printed.

`command`

`command [`*command* `[`*args* `...]]`

Runs *command* with *arg* ignoring shell functions. If you have a shell function called `ls`, and you wish to call the command `ls`, you can say "command ls".

`hash`

`hash [-r] [`*name*`]`

For each *name*, the full pathname of the command is determined and remembered. The `-r` option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented.

`local`

`local` *name*`[=`*value*`]`

Create a local variable called *name*, and give it *value*. `local` can only be used within a function; it makes the variable *name* have a visible scope restricted to that function and its children.

`readonly`

```
readonly [-f] [name ...]
```

The given *name*s are marked readonly and the values of these *name*s may not be changed by subsequent assignment. If the -f option is given, the functions corresponding to the *name*s are so marked. If no arguments are given, a list of all readonly names is printed.

`ulimit`

```
ulimit [-acdmstfpn [limit]]
```

Ulimit provides control over the resources available to processes started by the shell, on systems that allow such control. If an option is given, it is interpreted as follows:

`-a` all current limits are reported.

`-c` the maximum size of core files created.

`-d` the maximum size of a process's data segment.

`-m` the maximum resident set size.

`-s` the maximum stack size.

`-t` the maximum amount of cpu time in seconds.

`-f` the maximum size of files created by the shell.

`-p` the pipe buffer size.

`-n` the maximum number of open file descriptors.

If *limit* is given, it is the new value of the specified resource. Otherwise, the current value of the specified resource is printed. If no option is given, then `-f` is assumed. Values are in 1k increments, except for `-t`, which is in seconds, and `-p`, which is in increments of 512 bytes.

4.7 Bash Variables

`history_control`

Set to a value of "ignorespace", it means don't enter lines which begin with a SPC on the history list. Set to a value of "ignoredups", it means don't enter lines which match the last entered line. Unset, or any other value than those above mean to save all lines on the history list.

`HISTFILE` The name of the file that the command history is saved in.

`HISTSIZE` If set, this is the maximum number of commands to remember in the history.

`histchars`

Up to three characters which control history expansion, quick substitution, and tokenization. The first character is the *history-expansion-char*, that is, the character which signifies the start of a history expansion, normally '!'. The second character is the character which signifies 'quick substitution' when seen as the first character on a line, normally '^'. The optional third character is the character which signifies the remainder of the line is a comment, when found as the first character of a word, usually '#'.

`hostname_completion_file`

Contains the name of a file in the same format as '`/etc/hosts`' that should be read when the shell needs to complete a hostname. You can change the file interactively; the next time you want to complete a hostname Bash will add the contents of the new file to the already existing database.

`MAILCHECK`

How often (in seconds) that the shell should check for mail in the file(s) specified in `MAILPATH`.

`MAILPATH` Colon separated list of pathnames to check for mail in. You can also specify what message is printed by separating the pathname from the message with a '`?`'. `$_` stands for the name of the current mailfile. For example:

```
MAILPATH='/usr/spool/mail/bfox?"You have mail":~/shell-mail?"$_ has
mail!"'
```

`ignoreeof`

`IGNOREEOF`

Controls the action of the shell on receipt of an `EOF` character as the sole input. If set, then the value of it is the number of `EOF` characters that can be seen in a row as sole input characters before the shell will exit. If the variable exists but does not have a numeric value (or has no value) then the default is 10. if the variable does not exist, then `EOF` signifies the end of input to the shell. This is only in effect for interactive shells.

`auto_resume`

This variable controls how the shell interacts with the user and job control. If this variable exists then single word simple commands without redirects are treated as candidates for resumption of an existing job. There is no ambiguity allowed; if you have more than one job beginning with the string that you have typed, then the most recently accessed job will be selected.

`no_exit_on_failed_exec`

If this variable exists, the shell will not exit in the case that it couldn't execute the file specified in the `exec` command.

`PROMPT_COMMAND`

If present, this contains a string which is a command to execute before the printing of each toplevel prompt.

`nolinks` If present, says not to follow symbolic links when doing commands that change the current working directory. By default, bash follows the logical chain of directories when performing `cd` type commands.

For example, if '`/usr/sys`' is a link to '`/usr/local/sys`' then:

```
cd /usr/sys; echo $PWD -> /usr/sys
cd ..; pwd -> /usr
```

If `nolinks` is exists, then:

```
cd /usr/sys; echo $PWD -> /usr/local/sys
cd ..; pwd -> /usr/local
```

5 Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint. It should be considered a user's guide. For information on using the GNU History Library in your own programs, see ⟨undefined⟩ [Programming with GNU History], page ⟨undefined⟩.

5.1 History Interaction

The History library provides a history expansion feature that is similar to the history expansion in Csh. The following text describes the sytax that you use to manipulate the history information.

History expansion takes place in two parts. The first is to determine which line from the previous history should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is called the *event*, and the portions of that line that are acted upon are called *words*. The line is broken into words in the same fashion that the Bash shell does, so that several English (or Unix) words surrounded by quotes are considered as one word.

5.1.1 Event Designators

An event designator is a reference to a command line entry in the history list.

`!` Start a history subsititution, except when followed by a space, tab, or the end of the line... `=` or `(`.

`!!` Refer to the previous command. This is a synonym for `!-1`.

`!n` Refer to command line *n*.

`!-n` Refer to the command line *n* lines back.

`!string` Refer to the most recent command starting with *string*.

`!?string`[`?`]
Refer to the most recent command containing *string*.

5.1.2 Word Designators

A `:` separates the event specification from the word designator. It can be omitted if the word

designator begins with a `^`, `$`, `*` or `%`. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

`0 (zero)`	The zero'th word. For many applications, this is the command word.
`n`	The *n*'th word.
`^`	The first argument. that is, word 1.
`$`	The last argument.
`%`	The word matched by the most recent `?string?` search.
`x-y`	A range of words; `-`*y* Abbreviates `0-`*y*.
`*`	All of the words, excepting the zero'th. This is a synonym for `1-$`. It is not an error to use `*` if there is just one word in the event. The empty string is returned in that case.

5.1.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`.

`#`	The entire command line typed so far. This means the current command, not the previous command, so it really isn't a word designator, and doesn't belong in this section.
`h`	Remove a trailing pathname component, leaving only the head.
`r`	Remove a trailing suffix of the form '`.`'*suffix*, leaving the basename.
`e`	Remove all but the suffix.
`t`	Remove all leading pathname components, leaving the tail.
`p`	Print the new command but do not execute it.

6 Command Line Editing

This text describes GNU's command line editing interface.

6.1 Introduction to Line Editing

In this tex a the following notation is used to describe 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 6.3 [Readline Init File], page 20, for more info).

6.2 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.

6.2.1 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.

6.2.2 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.

6.2.3 Readline Killing Commands

The act of *cutting* text means to delete the text from the line, and to save away the deleted text for later use, just as if you had cut the text out of the line with a pair of scissors. There is a

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

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

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

`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.

6.2.4 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`.

6.3 Readline Init File

Although the Readline library comes with a set of 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.

6.3.1 Readline Init Syntax

You can start up with a vi-like editing mode by placing

```
set editing-mode vi
```

in your '`~/.inputrc`' file.

You can have Readline use a single line for display, scrolling the input between the two edges of the screen by placing

```
set horizontal-scroll-mode On
```

in your '`~/.inputrc`' file.

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. Here is an example:

```
# This is a comment line.
Meta-Rubout:  backward-kill-word
Control-u:  universal-argument
```

6.3.1.1 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.

6.3.1.2 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 too 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 neccessary.

6.3.1.3 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 arg says to kill the characters instead of deleting them.

`quoted-insert (C-q, C-v)`
: Add the next character that you type to the line verbatim. This is how to insert 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 args don't work.

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

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

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

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

6.3.1.4 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)`
: Do what C-u used to do in Unix line input. We save the killed text on the kill-ring, though.

`unix-word-rubout (C-w)`
: Do what C-w used to do 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.

6.3.1.5 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 emacs. By default, this is not bound.

6.3.1.6 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.

6.3.1.7 Some Miscellaneous Commands

`abort (C-g)`
: Ding! Stops things.

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

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

6.3.2 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 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.

Appendix A Feature Index by Type

(Index is empty)

7 Feature Index by Concept

(Index is empty)

Table of Contents