

# BASH REFERENCE

## CONTENTS

Aliasing .....	6
Arithmetic Evaluation.....	8
Arrays.....	13
Brace Expansion.....	6
Built-In Commands .....	16
Command Line Arguments .....	3
Command Substitution .....	8
Conditional Expressions.....	15
Control Commands .....	14
Definitions.....	2
Execution Order .....	13
Field Splitting .....	8
Functions.....	12
History Substitution.....	5
Input/Output .....	13
Invocation and Startup .....	3
Job Ids and Job Control.....	24
Options To <b>set</b> .....	22
Options To <b>shopt</b> .....	23
Options To <b>test</b> .....	21
Patterns.....	9
Pre-Defined Variables .....	10
Process Substitution .....	8
Prompting .....	4
Quoting .....	6
Readline .....	25
Readline Directives .....	25
Readline Key Bindings.....	25
Readline Variables.....	26
Restricted <b>bash</b> .....	2
Signals and Traps.....	13
Special Characters.....	24
Tilde Substitution.....	6
Variable Assignment.....	9
Variable Names .....	9
Variable Substitution .....	7

This reference card was written by Arnold Robbins. We thank Chet Ramey (**bash**'s maintainer) for his help.

## OTHER SSC PRODUCTS:

**Specialized Systems Consultants, Inc.**  
(206)FOR-UNIX/(206)782-7733  
FAX: (206)782-7191  
E-mail: sales@ssc.com  
URL: http://www.ssc.com

**Linux Journal—The Premier Linux Magazine**  
**Technical Books and CDs**  
**SAMBA: Integrating UNIX and Windows**  
**Shell Tutorials, KSH Reference**  
**VI & Emacs References, VI Tutorial**

© Copyright 1999 Specialized Systems Consultants, Inc.,  
P.O. Box 55549, Seattle, WA 98155-0549.  
All Rights Reserved.

## DEFINITIONS

This card describes version 2.02.0 of **bash**.

Several typefaces are used to clarify the meaning:

- **Serifa Bold** is used for computer input.
- *Serifa Italic* is used to indicate user input and for syntactic placeholders, such as *variable* or *cmd*.
- **Serifa Roman** is used for explanatory text.

*blank* – separator between words. Blanks consist of one or more spaces and/or tab characters. In addition, words are terminated by any of the following characters:

**;&()|<>space tab newline**

*command* – a series of *words*.

*list* – one or more *pipelines*. Can be separated by **;**, **&**, **&&**, **||** and optionally be terminated by **;**, **&**.

*n* – an integer.

*name* – a variable, alias, function or command name.

*keyword* – a reserved word in the **bash** language. Keywords are special only after a **;** or newline, after another keyword, and in certain other contexts.

*pat* – a **bash** pattern. See Patterns.

*pipeline* – a command or multiple commands connected by a pipe (**|**).

*string* – a collection of characters treated as a unit.

*substitution* – the process of replacing parts of the command line with different text, e.g., replacing a variable with its value. **bash** performs many substitutions. This card lists them in the order they are performed.

*word* – a generic argument; a word. Quoting may be necessary if it contains special characters.

## RESTRICTED bash

If **bash** is invoked as **rbash**, or with the **-r** option, it is *restricted*. The following actions are not allowed in a restricted shell:

changing directory with **cd**  
setting or unsetting **\$SHELL** or **\$PATH**  
using path names for commands that contain /  
using a path name that contains / for the **.** command  
importing functions from the environment  
parsing **\$SHELLOPTS** at startup  
redirecting output with any of **>**, **>|**, **<>**, **>&**, **&>**, or **>>**  
using **exec** to run a different command  
adding or deleting built-in commands with **enable**  
using **command -p** to bypass a restricted **\$PATH**  
using **set +r** or **set +restricted**

These restrictions are in effect *after* executing all startup files, allowing the author of the startup files full control in setting up the restricted environment. (In practice, restricted shells are not used much, as they are difficult to set up correctly.)

## Error Reporting

If you find an error in this reference and are the first to report it, we will send you a free copy of any of our references. Please write, or send electronic mail to **bugs@ssc.com**.

## COMMAND LINE ARGUMENTS

**bash** accepts the one letter options to **set**, and the additional one letter and GNU-style long options shown below.

**\$ bash [options] [args]**

**-**  
**--**  
**-c cmd**  
**-D**  
**-i**  
**-r**  
**-s**  
**--dump-po-strings**  
**--dump-strings**  
**--help**  
**--login**  
**--noediting**  
**--noprofile**  
**--norc**  
**--posix**  
**--rcfile file**  
**--restricted**  
**--verbose**  
**--version**

ends option processing  
ends option processing  
execute *cmd* (default reads command from file named in first entry of *args* and found via path search)  
print all double quoted strings that are preceded by a **\$** to **stdout**. This implies **-n**, no commands are executed  
set interactive mode  
set restricted mode  
read commands from **stdin** (default)  
same as **-D**, but output in GNU **gettext** format  
same as **-D**  
display a help message and exit successfully  
act like a login shell  
do not use the **readline** library to read commands when interactive  
do not read any of the initialization files. See Invocation And Startup, below  
do not read **~/bashrc** if interactive. See Invocation And Startup, below  
follow the IEEE POSIX 1003.2 standard  
use *file* instead of **~/bashrc** if interactive  
same as **-r**  
same as **set -v**  
print version information on **stdout** and exit successfully

## INVOCATION AND STARTUP

There are five ways that **bash** runs: normal interactive, normal non-interactive, as **sh**, in POSIX mode, or invoked via **rshd**.

1. Normal interactive: Login shells run commands in **/etc/profile**. The first of **~/bash\_profile**, **~/bash\_login**, and **~/profile** that is found is executed. This stage is skipped if **--noprofile** is used.

Upon logout, **bash** runs **~/bash\_logout** if it exists.

Interactive non-login shells execute **~/bashrc**, if it exists. The **--rcfile file** option changes the file that is used.

2. Normal non-interactive: Non-interactive shells do variable, command, and arithmetic substitution on the *value* of **\$BASH\_ENV**, and if the result names an existing file, that file is executed.

## INVOCATION AND STARTUP (continued)

3. Invoked as **sh**: Interactive login shells read and execute **/etc/profile** and **~/profile** if they exist. These files are skipped if **--noprofile** is used. Interactive shells expand **\$ENV** and execute that file if it exists. Non-interactive shells do not read any startup files. After the startup files are executed, **bash** enters POSIX mode.

4. POSIX mode: When started with **--posix**, interactive shells expand **\$ENV** and execute the given file. No other startup files are read.

5. Invoked via **rshd**: If run from **rshd** and not invoked as **sh**, **bash** reads **~/bashrc**. The **--norc** option skips this step, and the **--rcfile** option changes the file, but **rshd** usually does not pass these options on to the shell it invokes.

If **\$SHELLOPTS** exists in the environment at startup, **bash** enables the given options.

## PROMPTING

When interactive, **bash** displays the primary and secondary prompt strings, **\$PS1** and **\$PS2**. **bash** expands the following escape sequences in the values of these strings.

<b>\a</b>	an ASCII BEL character (octal 07)
<b>\d</b>	the date in "Weekday Month Day" format
<b>\e</b>	an ASCII escape character (octal 033)
<b>\h</b>	the hostname up to the first dot (.)
<b>\H</b>	the full hostname
<b>\n</b>	a newline
<b>\r</b>	a carriage return
<b>\s</b>	the name of the shell (basename of <b>\$0</b> )
<b>\t</b>	the time in 24-hour HH:MM:SS format
<b>\T</b>	the time in 12-hour HH:MM:SS format
<b>\u</b>	the user's username
<b>\v</b>	the version of <b>bash</b> (e.g., 2.02)
<b>\V</b>	the version and patchlevel of <b>bash</b> (e.g., 2.02.0)
<b>\w</b>	the current working directory
<b>\W</b>	the basename of the current working directory
<b>\!</b>	the history number of this command
<b>\#</b>	the command number of this command
<b>\\$</b>	a # if the effective UID is 0, otherwise a <b>\$</b>
<b>\@</b>	the time in 12-hour am/pm format
<b>\@</b>	a backslash
<b>\nnn</b>	the character corresponding to octal value <i>nnn</i>
<b>\ </b>	start a sequence of non-printing characters
<b>\ </b>	end a sequence of non-printing characters

The history number is the number of the command in the history list, which may include commands restored from the history file. The command number is the number of this command starting from the first command run by the current invocation of the shell.

The default value of **PS1** is **"\s-\v\\$ "**.

## HISTORY SUBSTITUTION

History expansion is similar to **cs**'s. It is enabled by default in interactive shells. History expansion happens before the shell breaks the input into words, although quoting is recognized and quoted text is treated as one history "word".

History substitution is performed on history *events*, which consist of an *event designator* (which previous line to start with), a *word designator* (which word from that line to use, starting with zero), and one or more optional *modifiers* (which parts of the words to use). Colons separate the three parts, although the colon between the event designator and word designator may be omitted when the word designator begins with **^**, **\$**, **\***, **-**, or **%**. Each modifier is separated from the next one with a colon. The **histchars** variable specifies the start-of-history and quick substitution characters, and also the comment character that indicates that the rest of a line is a comment. The previous command is the default event if no event designator is supplied.

The event designators are:

```
!          start a history substitution
!n        command line n
!-n       current line minus n (n previous)
!!        the previous command
!str      most recent command line starting with str
!?str{?} most recent command line containing str
!#        the entire command line typed so far
~old~new  quick substitution: repeat last command changing old to new
```

The word designators are:

```
0         the zero'th word (command name)
n         word n
^         the first argument, i.e., word one
$         the last argument
%         the word matched by the most recent !?str? search
x-y      words x through y. -y is short for 0-y
*        words 1 through the last (like 1-$)
n*       words n through the last (like n-$)
n-       words n through the next to last
```

The modifiers are:

```
e         remove all but the suffix of a filename
g         make changes globally, use with s modifier, below
h         remove the last part of a filename, leaving the "head"
p         print the command but do not execute it
q         quote the generated text
r         remove the last suffix of a filename
s/old/new/ substitute new for old in the text. Any delimiter may be used. An & in the replacement means the value of old. With empty old, use last old, or the most recent !?str? search if there was no previous old
t         remove all but the last part of a filename, leaving the "tail"
x         quote the generated text, but break into words at blanks and newline
&        repeat the last substitution
```

## QUOTING

```
\c       quote single character c
`...'    old style command substitution
"..."   text treated as a single argument, double quotes removed; variable, command and arithmetic substitutions performed; use \ to quote $, \, `, and "
$"..."  like "...", but locale translation done
'...'    text treated as a single argument, single quotes removed; text between quotes left alone, cannot include `
${...}   text treated as a single argument, $ and single quotes removed; no substitutions performed; ANSI C and additional escape sequences processed:
\a       alert (bell)          \v       vertical tab
\b       backspace           \ddd     octal value ddd
\f       form feed           \xhhh    hex value hhh
\n       newline             \\       backslash
\r       carriage return     \e       escape, not in ANSI C
\t       horizontal tab
```

## ALIASING

```
alias name=value ...
```

Aliases are expanded when a command is read, not when executed. Alias names can contain any non-special character, not just alphanumerics, except for **=**. Alias expansion is done on the first *word* of a command. If the last character of the replacement text is a *blank*, then the next word in the command line is checked for alias expansion. Aliases can even be used to redefine shell keywords, but not in POSIX mode.

## BRACE EXPANSION

Brace expansion is similar to **cs**'s. A word must contain at least one unquoted left brace and comma to be expanded. **bash** expands the comma-separated items in order, the result is not sorted. Brace expansions may be nested. For example:

```
$ mkdir /usr/{gnu,local}/{src,bin,lib}
```

## TILDE SUBSTITUTION

```
~        substitute $HOME
~user    substitute user's home directory
~+       substitute $PWD
~-       substitute $OLDPWD
~n       substitute ${DIRSTACK[n]}. A leading + or - is allowed: negative values count from the end of the stack
```

Tilde substitution happens after alias expansion. It is done for *words* that begin with **~** and for variable assignment.

In variable assignments, it is also done after a **:** in the value. Tilde substitution is done as part of word expansion. This means for **\${name op word}**, *word* will be checked for tilde substitution, but only if the operation requires the value of the right-hand side.

## VARIABLE SUBSTITUTION

```
$(name)  reference to shell variable name
${name}  use braces to delimit shell variable name
${name-word} use variable name if set, else use word
${name=word} as above but also set name to word
${name?word} use name if set, otherwise print word and exit (interactive shells do not exit)
${name+word} use word if name is set, otherwise use nothing
${name[n]} element n in array name
${#name}  length of shell variable name
${#name[*]} number of elements in array name
${#name[@]} number of elements in array name
${name#pat} remove shortest leading substring of name that matches pat
${name##pat} remove longest leading substring of name that matches pat
${name%pat} remove shortest trailing substring of name that matches pat
${name%%pat} remove longest trailing substring of name that matches pat
${name: start} length characters of name starting at start (counting from 0); use rest of value if no length. Negative start counts from the end. If name is * or @ or an array indexed by * or @, start and length indicate the array index and count of elements. start and length can be arithmetic expressions
${name/pattern/string} value of name with first match of pattern replaced with string
${name/pattern} value of name with first match of pattern deleted
${name//pattern/string} value of name with every match of pattern replaced with string
${name/#pattern/string} value of name with match of pattern replaced with string; match must occur at beginning
${name/%pattern/string} value of name with match of pattern replaced with string; match occurs at end
```

**Note:** for **-**, **=**, **?**, and **+**, using *name*; instead of *name* tests whether *name* is set and non-NULL; using *name* tests only whether *name* is set.

For **#**, **##**, **%**, **%%**, **/**, **//**, **/#**, and **/%**, when *name* is **\*** or **@** or an array indexed by **\*** or **@**, the substring or substitution operation is applied to each element.

## ARITHMETIC EVALUATION

Arithmetic evaluation is done with the **let** built-in command, the **((...))** command and the **\$(...)** expansion for producing the result of an expression.

All arithmetic uses **long** integers. Use **typeset -i** to get integer variables. Integer constants look like **[base#]n** where *base* is a decimal number between two and 64, and *n* is in that base. The digits are **0-9**, **a-z**, **A-Z**, **\_** and **@**. A leading **0** or **0x** denote octal or hexadecimal.

The following operators based on C, with the same precedence and associativity, are available.

```
+ -      unary plus and minus
! ~      logical and bitwise negation
**       exponentiation (not in C)
* / %    multiply, divide, modulus
+ -      addition, subtraction
<< >>    left shift, right shift
< <= > >= comparisons
== !=    equals, not equals
&        bitwise AND
^        bitwise XOR
|        bitwise OR
&&       logical AND, short circuit
||       logical OR, short circuit
?        in-line conditional
= += -= *= /= %= &= |= ^= <<= >>= assignment operators
```

Inside **let**, **((...))**, and **\$(...)**, variable names do not need a **\$** to get their values.

## COMMAND SUBSTITUTION

```
$(command) new form
`command`  old form
```

Run *command*, substitute the results as arguments. Trailing newlines are removed. Characters in **\$IFS** separate words (see Field Splitting). The new form is preferred for simpler quoting rules.

```
$(expression) arithmetic substitution
```

The *expression* is evaluated, and the result is used as an argument to the current command.

## PROCESS SUBSTITUTION

```
cmd <(list1) >(list2)
```

Runs *list1* and *list2* asynchronously, with **stdin** and **stdout** respectively connected via pipes using **fifo**s or files in **/dev/fd**. These file names become arguments to *cmd*, which expects to read its first argument and write its second. This only works if you have **/dev/fd** or **fifo**s.

## FIELD SPLITTING

Quoted text becomes one word. Otherwise, occurrences of any character in **\$IFS** separate words. Multiple whitespace characters that are in **\$IFS** do not delimit empty words, while multiple non-whitespace characters do. When **\$IFS** is not the default value, sequences of leading and trailing **\$IFS** whitespace characters are removed, and printable characters in **\$IFS** surrounded by adjacent **\$IFS** whitespace characters delimit fields. If **\$IFS** is **NULL**, **bash** does not do field splitting.

## PATTERNS

<b>?</b>	match single character in filename
<b>*</b>	match 0 or more characters in filename
<b>[chars]</b>	match any of <i>chars</i>
	(pair separated by a <b>-</b> matches a range)
<b>[!chars]</b>	match any except <i>chars</i>
<b>[^chars]</b>	match any except <i>chars</i>

If the **extglob** option to **shopt** is set, the following extended matching facilities may be used.

<b>?(pat-list)</b>	optionally match any of the patterns
<b>*(pat-list)</b>	match 0 or more of any of the patterns
<b>+(pat-list)</b>	match 1 or more of any of the patterns
<b>@(pat-list)</b>	match exactly 1 of any of the patterns
<b>!(pat-list)</b>	match anything but any of the patterns

*pat-list* is a list of one or more patterns separated by |.

The POSIX **[[=c=]]** and **[.[c.]]** notations for same-weight characters and collating elements are accepted. The notation **[[:class:]]** defines character classes:

<b>alnum</b>	alphanumeric	<b>lower</b>	lower-case
<b>alpha</b>	alphabetic	<b>print</b>	printable
<b>blank</b>	space or tab	<b>punct</b>	punctuation
<b>cntrl</b>	control	<b>space</b>	whitespace
<b>digit</b>	decimal	<b>upper</b>	upper-case
<b>graph</b>	non-spaces	<b>xdigit</b>	hexadecimal

Three **shopt** options affect pattern matching.

<b>dotglob</b>	include files whose names begin with <b>.</b>
<b>nocaseglob</b>	ignore case when matching
<b>nullglob</b>	remove patterns that don't match

When expanding filenames, **.** and **..** are ignored, filenames matching the patterns in **\$GLOBIGNORE** are also ignored and a leading **.** must be supplied in the pattern to match filenames that begin with **.** However, setting **GLOBIGNORE** enables the **dotglob** option. Include **\*** in **GLOBIGNORE** to get the default behavior.

## VARIABLE NAMES

Variable names are made up of letters, digits and underscores. They may not start with a digit. There is no limit on the length of a variable name, and the case of letters is significant.

## VARIABLE ASSIGNMENT

Assignments to integer variables undergo arithmetic evaluation. Variable assignments have one of the following forms.

<b>name=word</b>	set <i>name</i> to <i>word</i>
<b>name[index]=word</b>	set element <i>index</i> of array <i>name</i> to <i>word</i>
<b>name=(word ...)</b>	set indexed array <i>name</i> to <i>words</i>
<b>name=[(num)=word ...]</b>	set given indices of array <i>name</i> to <i>words</i>

## PRE-DEFINED VARIABLES

<b>\$n</b>	use positional parameter <i>n</i> , $n \leq 9$
<b>\${n}</b>	use positional parameter <i>n</i>
<b>\$*</b>	all positional parameters
<b>@*</b>	all positional parameters
<b>"\$*"</b>	equivalent to <b>"\$1 \$2 ..."</b>
<b>"\$@"</b>	equivalent to <b>"\$1" "\$2" ...</b>
<b>\$#</b>	number of positional parameters
<b>\$-</b>	options to shell or by <b>set</b>
<b> \$?</b>	value returned by last command
<b> \$\$</b>	process number of current shell
<b> \$!</b>	process number of last background cmd
<b> \$_</b>	name of program in environment at startup. Value of last positional argument in last command. Name of changed mail file in <b>\$MAILPATH</b>

### \$auto\_resume

enables use of single-word commands to match stopped jobs for foregrounding. With a value of **exact**, the word must exactly match the command used to start the job. With a value of **substring**, the typed word can be a substring of the command, like **%?string**

<b>\$BASH</b>	full file name used to invoke <b>bash</b>
<b>\$BASH_ENV</b>	in normal non-interactive shells only, value is variable, command and arithmetic substituted for path of startup file (See Invocation And Startup)

<b>\$BASH_VERSION</b>	the version of <b>bash</b>
<b>\$BASH_VERSINFO[0]</b>	the major version number (release)
<b>\$BASH_VERSINFO[1]</b>	the minor version number (version)
<b>\$BASH_VERSINFO[2]</b>	the patchlevel
<b>\$BASH_VERSINFO[3]</b>	the build version
<b>\$BASH_VERSINFO[4]</b>	the release status
<b>\$BASH_VERSINFO[5]</b>	same as <b>\$MACHTYPE</b>

<b>\$CDPATH</b>	search path for <b>cd</b> command
<b>\$DIRSTACK[*]</b>	array variable containing the <b>pushd</b> and <b>popd</b> directory stack

<b>\$ENV</b>	in interactive POSIX mode shells, or when invoked as <b>sh</b> , value is variable, command and arithmetic substituted for path of startup file
--------------	---

<b>\$EUID</b>	the effective user id (readonly)
<b>\$FCEDIT</b>	default editor for the <b>fc</b> command (no default value)

<b>\$FGIGNORE</b>	colon-separated list of suffixes giving the set of filenames to ignore when doing filename completion using <b>readline</b>
-------------------	---

<b>\$GLOBIGNORE</b>	colon-separated list of patterns giving the set of filenames to ignore when doing pattern matching
---------------------	--

<b>\$GROUPS[*]</b>	readonly array variable with the list of groups the user belongs to
--------------------	---

<b>\$histchars</b>	characters that control <b>cs</b> -style history (default: <b>!#</b> ). See History Substitution
--------------------	--

## PRE-DEFINED VARIABLES (continued)

<b>\$HISTCMD</b>	history number of the current command
------------------	---------------------------------------

<b>\$HISTCONTROL</b>	with a value of <b>ignorespace</b> , do not enter lines that begin with spaces into the history file. With a value of <b>ignoredups</b> , do not enter a line that matches the previous line. Use <b>ignoreboth</b> to combine both options where command history is stored
----------------------	---

<b>\$HISTFILE</b>	file name where command history is stored
<b>\$HISTFILESIZE</b>	maximum number of lines to keep in <b>\$HISTFILE</b>

<b>\$HISTIGNORE</b>	colon-separated list of patterns; if the current line matches any of them, the line is not entered in the history file. <b>&amp;</b> represents the last history line. Patterns must match the whole line
---------------------	---

<b>\$HISTSIZE</b>	number of previous commands to keep available while <b>bash</b> is running
-------------------	--

<b>\$HOME</b>	home directory for <b>cd</b> command and value used for tilde expansion
---------------	---

<b>\$HOSTFILE</b>	file in format of <b>/etc/hosts</b> to use for hostname completion
-------------------	--

<b>\$HOSTNAME</b>	name of the current host
<b>\$HOSTTYPE</b>	string describing the current host

<b>\$IFS</b>	field separators ( <b>space</b> , <b>tab</b> , <b>newline</b> ) for interactive shells, the number of consecutive EOFs that must be entered before <b>bash</b> actually exits
<b>\$IGNOREEOF</b>	name of <b>readline</b> startup file, overrides <b>~/inputrc</b>

<b>\$INPUTRC</b>	name of <b>readline</b> startup file, overrides <b>~/inputrc</b>
------------------	--

<b>\$LANG</b>	name of current locale
<b>\$LC_ALL</b>	current locale; overrides <b>\$LANG</b> and other <b>\$LC</b> variables

<b>\$LC_COLLATE</b>	current locale for character collation, includes sorting results of filename expansion
---------------------	--

<b>\$LC_CTYPE</b>	current locale for character class functions (see Patterns)
-------------------	---

<b>\$LC_MESSAGES</b>	current locale for translating <b>"..."</b> strings
----------------------	---

<b>\$LINENO</b>	line number of line being executed in script or function
-----------------	--

<b>\$MACHTYPE</b>	a string in GNU <i>cpu-company-system</i> format describing the machine running <b>bash</b>
-------------------	---

<b>\$MAIL</b>	name of a mail file, if any
<b>\$MAILCHECK</b>	check for mail every <i>n</i> seconds ( <b>60</b> default)

<b>\$MAILPATH</b>	filenames to check for new mail; uses : separator; <i>filename</i> may be followed by <i>?message</i> ; <b>_</b> in <i>message</i> is matched mail file name. Overrides <b>\$MAIL</b>
-------------------	---

<b>\$OLDPWD</b>	previous working directory
<b>\$OPTARG</b>	value of last argument processed by <b>getopts</b>

<b>\$OPTERR</b>	if set to 1, display error messages from <b>getopts</b> (default: 1)
-----------------	--

<b>\$OPTIND</b>	index of last argument processed by <b>getopts</b>
-----------------	--

## PRE-DEFINED VARIABLES (continued)

<b>\$OSTYPE</b>	string describing the operating system running <b>bash</b>
-----------------	--

<b>\$PATH</b>	command search path
<b>\$PIPESTATUS[*]</b>	array variable containing exit status values from processes in the most recently executed foreground pipeline

<b>\$PPID</b>	process id of shell's parent
<b>\$PROMPT_COMMAND</b>	command to run before each primary prompt

<b>\$PS1</b>	primary prompt string ( <b>\s-\v\S</b> )
<b>\$PS2</b>	secondary prompt string ( <b>&gt;</b> )
<b>\$PS3</b>	<b>select</b> command prompt string ( <b>#!?</b> )

<b>\$PS4</b>	tracing prompt string ( <b>+</b> )
<b>\$PWD</b>	current working directory
<b>\$RANDOM</b>	set each time it's referenced, 0 - 32767

<b>\$REPLY</b>	set by the <b>select</b> and <b>read</b> commands
----------------	---

<b>\$SECONDS</b>	number of seconds since shell invocation
<b> \$\$HELL</b>	name of this shell
<b> \$\$HELLOPTS</b>	colon-separated list of the enabled shell options for <b>set -o</b>

<b> \$\$SHLVL</b>	incremented by one for each sub- <b>bash</b>
-------------------	--

<b>\$TIMEFORMAT</b>	format string for output of <b>time</b> keyword. Special constructs introduced by %.
---------------------	--

<b>%[p][l]R</b>	elapsed secs
<b>%[p][l]U</b>	user CPU secs
<b>%[p][l]S</b>	system CPU secs
<b>%P</b>	CPU percentage
<b>%%</b>	literal %

	Optional <i>p</i> gives the precision, the number of digits after the decimal point; it must be between 0 and 3. Optional <i>l</i> produces a longer format, in the form <b>MMmSS.FFs</b>
--	---

<b>\$TMOUT</b>	number of seconds to wait during prompt before terminating
----------------	--

<b>\$UID</b>	the real user id (readonly)
--------------	-----------------------------

## FUNCTIONS

Functions run in the same process as the calling script, and share the open files and current directory. They access their parameters like a script, via **\$1**, **\$2** and so on. **\$0** does not change. **return** may be used inside a function or **.** script. Functions share traps with the parent script, except for **DEBUG**. Functions may be recursive, and may have local variables, declared using **declare**, **local**, or **typeset**. Functions may be exported into the environment with **export -f**.

## INPUT/OUTPUT

Redirections are done left to right, after pipes are set up. Default file descriptors are **stdin** and **stdout**. File descriptors above 2 are marked close-on-exec.

```
&>ward      send stdout and stderr to ward
>>ward      send stdout and stderr to ward
[n]<file     use file for input
[n]>file     use file for output
[n]<|file    like >, but overrides noclobber
[n]>>file    like > but append to file if it exists
[n]<>file    open file for read/write (default: fd0)
[n]<&m       duplicate input file descriptor from m
[n]>&m       duplicate output file descriptor from m
[n]<&-       close input file descriptor
[n]>&-       close output file descriptor
[n]<<ward    input comes from the shell script; treat a line with
ward as EOF on input. If any of ward is quoted, no
additional processing is done on input by the shell.
Otherwise:
```

- do variable, command, arithmetic substitutions
- ignore escaped newlines
- use \ to quote \, \$, `, and first character of ward

```
[n]<<-ward  as above, but with leading tabs ignored
Of &> and >&, the first is preferred. It is equivalent to
>ward 2>&1.
```

## EXECUTION ORDER

All substitutions and I/O redirections are performed before a command is actually executed.

**bash** maintains an internal hash table for caching external commands. Initially, this table is empty. As commands are found by searching the directories listed in **\$PATH**, they are added to the hash table.

The command search order is shell functions first, built-in commands second, and external commands (first in the internal hash table, and then via **\$PATH**) third.

## SIGNALS AND TRAPS

Signal handling is done with the **trap** built-in command. The *ward* argument describing code to execute upon receipt of the signal is scanned twice by **bash**; once when the **trap** command is executed, and again when the signal is caught. Therefore it is best to use single quotes for the **trap** command. Traps are executed in order of signal number. You cannot change the status of a signal that was ignored when the shell started up.

Traps on **DEBUG** happen after commands are executed.

Backgrounded commands (those followed by **&**) will ignore the **SIGINT** and **SIGQUIT** signals if the **monitor** option is turned off. Otherwise, they inherit the values of the parent **bash**.

## ARRAYS

Arrays in **bash** have no limits on the number of elements. Array indices start at 0. Array subscripts can be arithmetic expressions. Array elements need not be contiguous. **bash** does not have associative arrays.

## CONTROL COMMANDS

```
! pipeline  execute pipeline. If exit status was non-zero, exit
zero. If exit status was zero, exit 1
case word in [(([pat 1|[pat 2]...)] list ;;)... esac
execute list associated with pat that matches word.
Field splitting is not done for word. pat is a bash
pattern (see Patterns). | is used to indicate an OR
condition. Use leading ( if case is inside $( )
for name [in words] ; do list ; done
sequentially assign each word to name and execute
list. If in words is missing use the positional
parameters
[function] func () { list ; }
define function func, body is list (see Functions)
if list1 ; then list2 ; elif list3 ; then list4 ;... ; else list5 ; fi
if executing list1 returns successful exit status,
execute list2 else ...
select name [in words] ; do list ; done
print a menu of words, prompt with $PS3 and read a
line from stdin, saving it in $REPLY. If the line is
the number of one of the words, set name to it,
otherwise set name to NULL. Execute list. If in
words is missing use the positional parameters.
bash automatically reprints the menu at the end of
the loop
time [-p] pipeline
execute pipeline; print elapsed, system and user
times on stderr.
-p print times in POSIX format
The $TIMEFORMAT variable controls the format of
the output if -p is not used. bash uses the value
$'\nreal%3lR\nuser%3lU\nsys%3lS' if there is
no value for $TIMEFORMAT
until list 1 ; do list 2 ; done
like while but negate the termination test
while list 1 ; do list 2 ; do ne
execute list 1. If last command in list 1 had a
successful exit status, execute list 2 followed by
list 1. Repeat until last command in list 1 returns an
unsuccessful exit status
((...))
arithmetic evaluation, like let "... "
[[expression]]
evaluate expression, return successful exit status if
true, unsuccessful if false (see Conditional
Expressions for details)
(list)
execute list in a sub-shell
{list;}
execute list in the current shell
```

## CONDITIONAL EXPRESSIONS

Used with the **[...]** compound command, which does not do pattern expansion or word splitting.

```
string      true if string is not NULL
-a file     true if file exists (-e is preferred)
-b file     true if file is a block device
-c file     true if file is a character device
-d file     true if file is a directory
-e file     true if file exists
-f file     true if file is a regular file
-g file     true if file has set gid bit set
-G file     true if file group is effective gid
-h file     true if file is a symbolic link
-k file     true if file has sticky bit set
-L file     true if file is a symbolic link
-n string   true if string has non-zero length
-N file     true if file exists and was modified since
last read
-o option   true if option is on
-O file     true if file owner is effective uid
-p file     true if file is a fifo (named pipe)
-r file     true if file is readable
-s file     true if file has non-zero size
-S file     true if file is a socket
-t filedes true if filedes is a terminal
-u file     true if file has set uid bit set
-w file     true if file is writable
-x file     true if file is executable
-z string   true if string has zero length
file1 -nt file2
true if file1 is newer than file2 or file2
does not exist
file1 -ot file2
true if file1 is older than file2 or file2
does not exist
file1 -ef file2
true if file1 and file2 are the same file
string == pattern
true if string matches pattern
string != pattern
true if string does not match pattern
string1 < string2
true if string1 is before string2
string1 > string2
true if string1 is after string2
exp1 -eq exp2 true if exp1 equals exp2
exp1 -ne exp2 true if exp1 does not equal exp2
exp1 -lt exp2 true if exp1 is less than exp2
exp1 -gt exp2 true if exp1 is greater than exp2
exp1 -le exp2 true if exp1 is less than or equal to exp2
exp1 -ge exp2 true if exp1 is greater than or
equal to exp2
(expression) true if expression is true, for grouping
! expression true if expression is false
exp1 && exp2 true if exp1 AND exp2 are true
exp1 || exp2 true if exp1 OR exp2 is true
```

If file is **/dev/fd/n**, then, if there is no **/dev/fd** directory, file descriptor *n* is checked. Otherwise, the real **/dev/fd/n** file is checked. Linux, FreeBSD, BSD/OS (and maybe others) return info for the indicated file descriptor, instead of the actual **/dev/fd** device file.

Both **&&** and **||** are short circuit. Operands of comparison operators undergo arithmetic evaluation. For **==** and **!=**, quote any part of *pattern* to treat it as a string.

## BUILT-IN COMMANDS

These commands are executed directly by the shell. Almost all accept **--** to mark the end of options.

```
. file
source file
read and execute commands from file. If
arguments, save and restore positional params.
Search $PATH; if nothing found, look in the current
directory
: null command; returns 0 exit status
[ see test
alias [-p] [name[=value] ...]
create an alias. With no arguments, print all
aliases. With name, print alias value for name
-p print alias before each alias
bg [jobid]
put jobid in the background
bind [-m map] [-lpPsSvV]
bind [-m map] [-q func] [-r keyseq] [-u func]
bind [-m map] -f file
bind [-m map] keyseq:func
display and/or modify readline function and key
bindings. The syntax is same as for ~/.inputrc
-f file read new bindings from file
-l list the names of all readline functions
-m map use the keymap map
-p list readline functions and bindings
for re-reading
-P list readline functions and bindings
-Q func show which keys invoke func
-r keyseq remove bindings for keyseq
-s list readline key sequences and macros
for re-reading
-S list readline key sequences and macros
-u func remove key bindings for func
-v list readline variable names and values
for re-reading
-V list readline variable names and values
break [n]
exit from enclosing for, while, until or select loop.
If n is supplied, exit from n'th enclosing loop
builtin shell-builtin [ args ...]
execute shell-builtin with given args and return
status. Useful for the body of a shell function that
redefines a built-in, e.g., cd
cd [-LP] [dir]
change current directory to dir ($HOME default).
Do directory path search using value of $CDPATH
-L use logical path for cd .., $PWD (default)
-P use physical path for cd .., $PWD
If both are given, the last one on the command line
wins
cd [-LP] -
change current directory to $OLDPWD
command [-pvV] name [arg ...]
without -v or -V, execute name with arguments arg
-p use a default search path, not $PATH
-v print a one word description of name
-V print a verbose description of name
continue [n]
do next iteration of enclosing for, while, until or
select loop. If n is supplied, iterate n'th enclosing
loop
```

## BUILT-IN COMMANDS (continued)

**declare** [**±affix**] [**-p**] [*name=value*]  
**typeset** [**±affix**] [**-p**] [*name=value*]  
 set attributes and values of variables. Inside functions, create new copies of the variables. Using + instead of - turns attributes off. With no names or attributes, print every variable's name and attributes  
 -a *name* is an array  
 -f each *name* is a function  
 -F don't show function definitions (bodies)  
 -i *name* is an integer; arithmetic evaluation is done upon assignment  
 -r mark *names* **readonly**  
 -x mark *names* for **export**

**dirs** [**-clpv**] [**+n**] [**-n**]  
 display the directory stack  
 +n show *n*'th entry from left,  $n \geq 0$   
 -n show *n*'th entry from right,  $n \geq 0$   
 -c clear the directory stack  
 -l print a longer format listing  
 -p print the stack one entry per line  
 -v print the stack one entry per line, with index numbers

**disown** [**-ar**] [**-h**] [*job ...*]  
 with no options, remove named *jobs* from the table of active jobs  
 -a remove or mark (with **-h**) all jobs  
 -h mark each *job* to not receive a **SIGHUP** when **bash** terminates  
 -r use with **-h** to mark just running jobs

**echo** [**-eEn**] [*words*]  
 echo *words*; -- is not special  
 -e expand \-escapes (see *echo(1)*)  
 -E never expand \-escapes  
 -n don't output trailing newline  
**printf** is more portable

**enable** [**-adnps**] [**-f file**] [*name ...*]  
 enable and disable shell built-ins, or load and unload new built-ins from shared library files. Disabling a built-in allows use of a disk file with the same name as a built-in  
 -a print all built-ins, with their status  
 -d delete a built-in loaded with **-f**  
 -f *file* load a new built-in *name* from *file*  
 -n disable *name*, or print disabled built-ins with no *names*  
 -p print enabled built-ins  
 -s print only POSIX special built-ins

**eval** [*words*]  
 evaluate *words* and execute result

**exec** [**-a name**] [**-cl**] [*words*]  
 execute *words* in place of the shell. If redirections only, change the shell's open files  
 -a use *name* for **argv[0]**  
 -c clear the environment first  
 -l place a - on **argv[0]** (like *login(1)*)  
 If the **exec** fails, non-interactive shells exit, unless the **shopt** option **execfail** is set

**exit** [*n*]  
 exit with return value *n*. Use **\$?** if no *n*

## BUILT-IN COMMANDS (continued)

**export** [**-fnp**] [*name=value ...*]  
 with no arguments, print names and values of exported variables. Otherwise, export *names* to the environment of commands  
 -f *names* refer to functions  
 -n stop exporting each *name*  
 -p print **export** before each variable

**fc** [**-e editor**][**-nlr**][*first [last]*]  
 print a range of commands from *first* to *last* from last  
**\$HISTSIZE** commands  
 -e run *editor* if supplied; if not, use first of **\$FCEDIT**, **\$EDITOR**, or **vi** on commands; execute result(s)  
 -l list on standard output instead of editing  
 -n don't print line numbers  
 -r reverse order of commands

**fc** -s [*old=new*] [*command*]  
 substitute *new* for *old* in *command* (or last command if no *command*) and execute the result

**fg** [*jobid*]  
 put *jobid* in the foreground

**getopts** *optstring name [arg ...]*  
 parse parameters and options (see *bash(1)*)

**hash** [**-r**] [**-p file**] [*name*]  
 with no arguments, print the hash table contents, giving hit count and file name  
 -p *file* enter *file* for *name* in the hash table  
 -r clear the internal hash table  
 Assignment to **\$PATH** also clears the hash table

**help** [*pattern*]  
 print help. With *pattern*, print help about all the commands that match *pattern*

**history** [*n*]  
**history -anrw** [*file*]  
**history** [**-c**]  
**history -p arg** [...]   
**history -s arg** [...]  
 with no options, print the command history. An argument of *n* prints only *n* lines. If supplied, use *file* instead of **\$HISTFILE**  
 -a append new history lines to history file  
 -c clear the history list  
 -n read new history lines in the file into the internal history list  
 -p perform history substitution and print the results  
 -r replace internal history with contents of history file  
 -s place the *args* into the history list for later use  
 -w write the internal history to the file

**jobs** [**-lnprs**] [*jobid ...*]  
**jobs -x command [args ...]**  
 list information about jobs  
 -l also list process id  
 -n only list stopped or exited jobs  
 -p only list process groups  
 -r only list running jobs  
 -s only list stopped jobs  
 -x replace any *jobid* in the command line with the corresponding process group ID, and execute the command

## BUILT-IN COMMANDS (continued)

**kill** [**-sig**] *jobid ...*  
**kill** [**-s signal**] [**-n signal**] *jobid ...*  
 send **SIGTERM** or given signal to named *jobids*. Signals are names listed in **/usr/include/signal.h** with or without the prefix "SIG". Stopped jobs get a **SIGCONT** first if *sig* is either **SIGTERM** or **SIGHUP**

**kill** -l [*sigs ...*]  
 list signal names and/or numbers. If *sig* is a numerical exit status, print the signal that killed the process

**let** *arg ...*  
 evaluate each *arg* as an arithmetic expression; exit 0 if the last expression was non-zero, 1 otherwise (see Arithmetic Evaluation)

**local** [*name=value ...*]  
 create variables with the given values local to a function. With no operands, print a list of local variables. Must be used inside a function

**logout**  
 exit a login shell

**popd** [**-n**] [**+n**] [**-n**]  
 remove entries from the directory stack. With no arguments, remove the top entry and **cd** there  
 +n remove *n*'th entry from left,  $n \geq 0$   
 -n remove *n*'th entry from right,  $n \geq 0$   
 -n don't change directory

**printf** *format [arg ...]*  
 print output like ANSI C **printf**, with extensions  
 %b expand escape sequences in strings  
 %q print quoted string that can be re-read  
 Format conversions are reused as needed

**pushd** [**-n**] [*dir*]  
**pushd** [**-n**] [**+n**] [**-n**]  
 add an entry to the directory stack. With no arguments, exchange the top two entries  
 +n rotate the stack so that the *n*'th entry from left is at the top,  $n \geq 0$   
 -n rotate the stack so that the *n*'th entry from right is at the top,  $n \geq 0$   
 -n don't change directory  
*dir* push *dir* on the stack and **cd** there

**pwd** [**-LP**]  
 print working directory name  
 -L print logical path (default)  
 -P print physical path  
 If both are given, the last one on the command line wins

**read** [**-a name**] [**-er**] [**-p prompt**] [*names ...*]  
 read **stdin** and assign to *names*. **\$IFS** splits input. **\$REPLY** is set if no *name* given. Exit 0 unless end-of-file encountered  
 -a read words into indexed array *name*  
 -e use **readline** if reading from a terminal  
 -p print *prompt* if reading from a terminal before reading  
 -r \ at end of line does not do line continuation

## BUILT-IN COMMANDS (continued)

**readonly** [**-afp**] [*name=value ...*]  
 mark *names* read-only; print list if no *names*  
 -a each *name* must be an array  
 -f each *name* must be a function  
 -p print **readonly** before each variable

**return** [*n*]  
 exit function or . script with return value *n*. With no *n*, return status of last command. If not in function or . script, print an error message

**set** [**-options**] [**-o option**] [*words*]  
 set flags and options (see Options To **set**). *words* set positional parameters

**set** [**+options**] [**+o option**] [*words*]  
 unset flags and options

**shift** [*n*]  
 rename positional parameters;  $\$n+1=\$1 ...$   
*n* defaults to 1

**shopt** [**-opqsu**] [*option ...*]  
 print or change values of shell options. With no arguments, print shell option information  
 -o only change **set -o** options  
 -p print settings for re-reading  
 -q quiet mode; exit status indicates option status  
 -s set (enable) given option; with no options, print those that are set  
 -u unset (disable) given option; with no options, print those that are unset  
 (See Options To **shopt**)

**suspend** [**-f**]  
 suspend the shell until **SIGCONT** is received  
 -f force suspension, even for login shell

**test**  
 evaluate conditional expressions (see Options To **test** and Conditional Expressions)

**times**  
 print accumulated process times

**trap** [**-lp**] [*ward*] [*sigs*]  
 execute *ward* if signal in *sigs* received. *sigs* are numbers or signal names with or without "SIG". With no *ward* or *sigs*, print traps. With no *ward*, reset *sigs* to entry defaults. If *ward* is "-", reset *sigs* to entry defaults. If *ward* is the null string, ignore *sigs*. If *sigs* is 0 or **EXIT**, execute *ward* on exit from shell. If *sigs* is **DEBUG**, run *ward* after every command.  
 -l print a list of signal names and numbers  
 -p print traps with quoting

**type** [**-apt**] *name ...*  
 describe how the shell interprets *name*  
 -a print all possible interpretations of *name*  
 -p print the name of the file to execute if *name* is an external program  
 -t print a keyword describing *name*

## BUILT-IN COMMANDS (continued)

**ulimit** [*type*] [*options*] [*limit*]  
set or print per-process limits  
*type* (default is both):  
-H hard limit  
-S soft limit

*options*:

- a all (display only)
- c core file size
- d "k" of data segment
- f maximum file size
- m "k" of physical memory
- n maximum file descriptor + 1
- p size of pipe buffers
- s "k" of stack segment
- t cpu seconds
- u max processes for one user
- v "k" of virtual memory

-f is assumed if no options are given. The size for -p is in 512-byte blocks; the others are in sizes of 1024 bytes

**umask** [-pS] [*mask*]  
set file creation permissions mask to complement of *mask* if octal, or symbolic value as in **chmod**. With no arguments, print current mask. An octal mask is permissions to remove, a symbolic mask is permissions to keep  
-p print output for re-reading  
-S print current mask in symbolic form

**unalias** [-a] [*names*]  
remove aliases *names*  
-a remove all aliases

**unset** [-fv] [*names*]  
unset variables *names* (same as -v)  
-f unset functions *names*  
-v unset variables *names*  
Unsetting **LINEENO**, **MAILCHECK**, **OPTARG**, **OPTIND**, **RANDOM**, **SECONDS**, **TMOU**T and \_ removes their special meaning, even if used afterwards

**wait** [*jobid* ...]  
wait for job *jobid*; if no *job*, wait for all children

## OPTIONS TO test

The **test** command, and its synonym [...], are built-in to **bash**. The command accepts all of the options listed in the Conditional Expressions section. However, since it is a command, options and arguments must be quoted to get proper behavior, and normal pattern expansion and field splitting are done. Parentheses used for grouping must be quoted. Arithmetic expansion is not done for numeric operators, and pattern matching is not done for == and !=. **test** complies with POSIX.

The -a and -o options have the following meanings, instead of the ones listed in Conditional Expressions:

-a logical AND  
-o logical OR

## OPTIONS TO set

The **set** command is complicated. Here is a summary. Use + instead of - to turn options off. With no arguments, **set** prints the names and values of all variables.

**set** [*±abBcEfhHkMnpPt uvx*] [*±o option* ...] [*arg* ...]

- a automatically export variables upon assignment
- b print job completion messages immediately, don't wait for next prompt
- B enable brace expansion (default)
- C force >| to overwrite for existing files
- e exit upon non-zero exit from a command
- f disable pattern expansion
- h save command locations in the internal hash table (default)
- H enable !-style history (default)
- k place all variable assignments in the environment (obsolete)
- m run background jobs in their own process group, print a message when they exit; set automatically for interactive shells on job control systems
- n read commands without executing them (ignored if interactive)
- o set options; with no arguments, print current settings
- allexport same as -a
- braceexpand same as -B
- emacs use an emacs-style line editor (default)
- errexit same as -e
- hashall same as -h
- histexpand same as -H
- history enable history
- ignoreeof like IGNOREEOF=10
- keyword same as -k
- monitor same as -m
- noclobber same as -C
- noexec same as -n
- noglob same as -f
- notify same as -b
- nounset same as -u
- onecmd same as -t
- physical same as -P
- posix obey the POSIX 1003.2 standard
- privileged same as -p
- verbose same as -v
- vi use a vi-style line editor
- xtrace same as -x

-p don't read \$ENV, do not take shell functions from environment, and ignore options in \$SHELLOPTS environment variable

-P follow the physical directory structure for commands that change the directory

-t read and execute one command, then exit

-u make it an error to substitute an unset variable

-v print input lines as they're read

## OPTIONS TO set (continued)

- x print commands as they're executed, preceded by expanded value of \$PS4. Output is quoted for later reuse
- turn off -v, -x, stop looking for flags; any remaining args set the positional parameters
- do not change flags; set positional parameters from argument list; with no args, unset the positional parameters

## OPTIONS TO shopt

The **shopt** command sets or unsets a number of options that affect how **bash** behaves. This section describes each option's effect when enabled. Unless noted, they are all disabled by default.

**cdable\_vars**  
treat an argument to **cd** that is not a directory as a variable whose value is the directory name

**cdspell**  
attempt to correct minor spelling errors in arguments to **cd**. Errors tried are transposed characters, a missing character or an extra character. Only obeyed in interactive shells

**checkhash**  
check that a command in the hash table still exists before trying to execute it. If it doesn't, search \$PATH

**checkwinsize**  
check the window size after each command and update \$LINES and \$COLUMNS

**cmdhist**  
attempt to save all lines of a multi-line command in the history file as one line, for easy re-editing

**dotglob**  
include files whose names begin with . in path expansions

**execfail**  
keep non-interactive shells from exiting when **exec** fails

**expand\_aliases**  
expand aliases as described in Aliases. Enabled automatically in interactive shells

**extglob**  
enable the extended pattern matching facilities (see Patterns)

**histappend**  
append the current history to \$HISTFILE upon exit, instead of overwriting it

**histreedit**  
if using **readline** and a history substitution fails, the user can re-edit the line

**histverify**  
if using **readline**, load the results of history substitution into **readline** for further editing

**hostcomplete**  
if using **readline**, attempt host completion on word containing @

**huponexit**  
send SIGHUP to all jobs when **bash** exits

**interactive\_comments**  
in interactive shells, a word starting with # starts a comment. Enabled by default

## OPTIONS TO shopt (continued)

**lithist**  
if **cmdhist** is also enabled, save multi-line commands with newlines, not semi-colons

**mailwarn**  
print a warning message if a file being checked for mail was accessed since the last time it was checked

**nocaseglob**  
do a case-insensitive match when expanding pathnames

**nullglob**  
remove patterns that don't match any file, instead of leaving them unchanged in the command line

**promptvars**  
do parameter expansion on the prompt variables before printing them. Enabled by default

**shift\_verbose**  
print an error message when the shift count is greater than the number of positional parameters

**sourcepath**  
use \$PATH to find shell files given to the . and source commands. Enabled by default

## SPECIAL CHARACTERS

# start of comment; terminated by newline  
| (pipe) connects two commands  
; command separator  
& run process in background; default **stdin** from /dev/null if no job control  
&& only run following command if previous command completed successfully  
|| only run following command if previous command failed  
' enclose string to be taken literally  
" enclose string to have variable, command and arithmetic substitution only  
\$( in-line command substitution (new style)  
` in-line command substitution (old style)  
{(...) arithmetic evaluation, like **let** "..."  
\$(...) in-line arithmetic evaluation  
\ treat following character literally  
\ **newline** line continuation

## JOB IDS AND JOB CONTROL

Jobs can be represented as follows:

*jobid* the job identifier for a job, where:

- % current job
- %+ current job
- %- previous job
- %?str job uniquely identified by *str*
- %n job number *n*
- %pref job whose command line begins with *pref*

Usually, a process ID may be used instead of a *jobid*. Commands that take a *jobid* use the current job if no *jobid* is supplied.

Traps on SIGCHLD execute whenever a job completes.

The commands **fg** and **bg** are only available on systems that support job control. This includes Linux, BSD systems, System V Release 4, and most UNIX systems.

## READLINE

The **readline** library implements command line editing. By default, it provides an *emacs* editing interface, although a *vi* interface is available. **readline** is initialized either from the file named by **\$INPUTRC** (if set) or from **~/inputrc**. In that file, you can use conditionals, define key bindings for macros and functions, and set variables.

From the **bash** level, the **bind** command allows you to add, remove and change macro and key bindings. There are five input mode map names that control the action taken for each input character. The map names are **emacs**, **emacs-standard**, **emacs-meta**, **emacs-ctlx**, **vi**, **vi-command**, and **vi-insert**. **emacs** is the same as **emacs-standard**, and **vi** is the same as **vi-command**.

You choose which editor you prefer with **set -o emacs** or **set -o vi** in your **~/bashrc** file, or at runtime.

**readline** understands the character names *DEL*, *ESC*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *RUBOUT*, *SPACE*, *SPC* and *TAB*.

## READLINE DIRECTIVES

Directives in the **inputrc** file provide conditional and include facilities similar to the C preprocessor.

```
$include      include a file, e.g., a system-wide /etc/inputrc file
$if          start a conditional, for terminal or application
              specific settings. You can test the following:
application= test the application, e.g. bash or gdb
mode=        test the editing mode, emacs or vi
term=       test the terminal type

The use of application= is optional; e.g., $if Bash
$else       start the "else" part of a conditional
$endif     finish a conditional
```

## READLINE KEY BINDINGS

Keys bound to a macro place the macro text into the input; keys bound to a function run the function.

You can use these escape sequences in bindings:

<b>\a</b>	alert (bell)	<b>\r</b>	carriage return
<b>\b</b>	backspace	<b>\t</b>	horizontal tab (TAB)
<b>\C-</b>	control prefix	<b>\v</b>	vertical tab
<b>\d</b>	delete (DEL)	<b>\</b>	backslash
<b>\e</b>	escape (ESC)	<b>\"</b>	literal "
<b>\f</b>	form feed	<b>\'</b>	literal '
<b>\M-</b>	meta prefix	<b>\ddd</b>	octal value <i>ddd</i>
<b>\n</b>	newline	<b>\xhhh</b>	hex value <i>hhh</i>

Macros and function bindings look like:

```
macro:      key-seq:"text"
function:   key-seq:function-name
```

Macros have quoted text on the right of the colon; functions have function names. A *key-seq* is either a single character or character name (such as **Control-o**), or a quoted string of characters (single or double quotes).

## READLINE VARIABLES

Variables control different aspects of **readline**'s behavior. You set a variable with

```
set variable value
```

Unless otherwise noted, *value* should be either **On** or **Off**. The descriptions below describe the effect when the variable is **On**. Default values are shown in parentheses.

```
bell-style (audible)
  defines how readline should ring the bell:
audible           ring the bell
no ne            never ring the bell
visible          flash the screen

comment-begin (#)
  insert this string for readline-insert-comment,
  (bound to M-# in emacs mode and to # in vi mode)

completion-ignore-case (Off)
  ignore case when doing completions

completion-query-items (100)
  if the number of completion items is less than this
  value, place them in the command line. Otherwise,
  ask the user if they should be shown

convert-meta (On)
  treat characters with the eighth bit set as the meta
  version of the equivalent seven bit character

disable-completion (Off)
  do not do completion

editing-mode (emacs)
  set the initial editing mode. Possible values are
emacs or vi

enable-keypad (Off)
  attempt to enable the application keypad. This may
  be needed to make the arrow keys work

expand-tilde (Off)
  attempt tilde expansion as part of word completion

input-meta (Off)
meta-flag (Off)
  enable eight bit input. The two variable names are
  synonyms

keymap (emacs)
  set the current keymap. See Readline for a list of
  allowed values. The editing-mode variable also
  affects the keymap

mark-directories (On)
  append a / to completed directory names

mark-modified-lines (Off)
  place a * at the front of modified history lines

output-meta (Off)
  print characters with the eighth bit set directly, not
  as M-x

print-completions-horizontally (Off)
  display completions horizontally, with the matches
  sorted alphabetically, instead of vertically down the
  screen

show-all-if-ambiguous (Off)
  immediately list words with multiple possible
  completions, instead of ringing the bell

visible-stats (Off)
  when listing possible completions, append a
  character that denotes the file's type
```

More information about **readline** can be found on-line at <http://www.ssc.com/ssc/bash>.