bash – GNU Bourne-Again SHell [options] [file] Copyright © 1989, 1991 by the Free Software Foundation, Inc. is an **sh**–compatible command language interpreter that executes commands read from the standard input or from a file. also incorporates useful features from the *Korn* and *C* shells (**ksh** and **csh**). is ultimately intended to be a faithful implementation of the IEEE Posix Shell and Tools specification (IEEE Working Group 1003.2). In addition to the single–character shell options documented in the description of the **set** builtin command, **bash** interprets the following flags when it is invoked: If the flag is present, then commands are read from If the flag is present, the shell is If the flag is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell. A single signals the end of options and disables further option processing. Any arguments after the are treated as filenames and arguments. An argument of is equivalent to an argument of **–**. also interprets a number of multi–character options. These options must appear on the command line before the single–character options to be recognized. Do not load the personal initialization file if the shell is interactive. This is the default if the shell name is Do not read either */etc/profile* or By default, normally reads these files when it is invoked as a login shell. **–rcfile** *file* Execute commands from instead of the standard personal initialization file if the shell is interactive. Show the version number of this instance of when starting. Do not be verbose when starting up (do not show the shell version or any other information). Make act as if it had been invoked by login(1). Do not perform curly brace expansion a la Do not use the GNU library to read command lines if interactive. If arguments remain after option processing, and neither the nor the option has been supplied, the first argument is assumed to be the name of a file containing shell commands. If is invoked in this fashion, is set to the name of the file, and the positional parameters are set to the remaining arguments. reads and executes commands from this file, then exits. A space or tab. A sequence of characters considered as a single unit by the shell. Also known as a A consisting only of alphanumeric characters and underscores, and beginning with an alphabetic character or an underscore. Also referred to as an A character that, when unquoted, separates words. One of the following:

| **& ; ( )** < > <**space**> <**tab**> A *token* that performs a control function. It is one of the following symbols: || **& && ; ;; ( )** | <**newline**> *Reserved words* are words that have a special meaning to the shell. The following words are recognized as reserved when unquoted and either the first word of a simple command (see below) or the third word of a or command: ! case do done elif else esac fi for function if in then until while { } A *simple command* is a sequence of optional variable assignments followed by *blank*–separated words and redirections, and terminated by a *control operator*. The first word specifies the command to be executed. The remaining words are passed as arguments to the invoked command. The return value of a *simple command* is its exit status, or 128+*n* if the command is terminated by signal A *pipeline* is a sequence of one or more commands separated by the character The format for a pipeline is: [!] *command* [| *command2* ...] The standard output of is connected to the standard input of This connection is performed before any redirections specified by the command (see below). If the reserved word precedes a pipeline, the exit status of that pipeline is the logical NOT of the exit status of the last command. Otherwise, the status of the pipeline is the exit status of the last command. The shell waits for all commands in the pipeline to terminate before returning a value. Each command in a pipeline is executed as a separate process (i.e. in a subshell). A *list* is a sequence of one or more pipelines separated by one of the operators or and optionally terminated by one of or Of these list operators, has highest precedence. has the next highest precedence, followed by and which have equal precedence. If a command is terminated by the control operator the shell executes the command in the *background* in a subshell. The shell does not wait for the command to finish. Commands separated by a are executed sequentially; the shell waits for each command to terminate in turn. The control operators and denote AND lists and OR lists, respectively. An AND list has the form *command* **&&** *command2* is executed if, and only if, returns an exit status of zero. An OR list has the form *command* || *command2* is executed if and only if returns a non-zero exit status. A *compound command* is one of the following: (*list*) *list* is executed in a subshell. Variable assignments and builtin commands that affect the shell's environment do not remain in effect after the command completes. { *list*; } *list* is simply executed in the current shell environment. This is known as a *group command*. **for** *name* [**in** *word*] **do** *list* **done** The list of words following **in** is expanded, generating a list of items. The variable *name* is set to each element of this list in turn, and *list* is executed each time. If the **in** *word* is omitted, the **for** command executes *list* once for each positional parameter that is set (see below). The exit status is the exit status of the last command, or zero if no commands were executed. **case** *word* **in** [*pattern* [| *pattern*] ...) *list* ;;] ... **esac** A **case** command first expands *word*, and tries to match it against each *pattern* in turn. When a match is found, the corresponding *list* is executed. After the first match, no subsequent matches are attempted. The exit status is zero if no patterns are matches. Otherwise, it is the exit status of the last command executed in *list*. **if** *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi** The is executed. If its exit status is zero, the **then** *list* is executed. Otherwise, each **elif** *list* is executed in turn, and if its exit status is zero, the corresponding **then** *list* is executed and the command completes. Otherwise, the **else** *list* is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true. **while** *list* **do** *list* **done until** *list* **do** *list* **done** The **while** command continuously executes the **do** *list* as long as the last command in *list* returns an exit status of zero. The **until** command is identical to the **while** command, except that the test is negated; the is executed as long as the last command in returns a non-zero exit status. The exit status of the **while** and **until** commands is the exit status of the last **do** *list* command executed, or zero if none was executed. [**function**] *name* () { *list*; } This defines a function named *name*. The *body* of the function is the of commands between { and }. This list is executed whenever *name* is specified as the name of a simple command. The exit status of a function is the exit status of the last command executed in the body. (See below.) In

a non–interactive shell, a word beginning with causes that word and all remaining characters on that line to be ignored. *Quoting* is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion. Each of the *metacharacters* listed above under has special meaning to the shell and must be quoted if they are to represent themselves. There are three quoting mechanisms: the escape character, single quotes, and double quotes. A non-quoted backslash (\) is the It preserves the literal value of the next character that follows, with the exception of <newline>. If a \<newline> pair appears, it is treated as a line continuation (that is, it is effectively ignored), if the backslash is non-quoted. Enclosing characters in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash. Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of and The characters and retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: ", or A double quote may be quoted within double quotes by preceding it with a backslash. The special parameters and have special meaning when in double quotes (see below). A is an entity that stores values, somewhat like a variable in a conventional programming language. It can be a a number, or one of the special characters listed below under For the shell's purposes, a is a parameter denoted by a A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the builtin command (see below). A may be assigned to by a statement of the form *name*=[*value*] If is not given, the variable is assigned the null string. All undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal. If the variable has its attribute set (see below in then is subject to arithmetic expansion even if the $[...] syntax does not appear. Word splitting is not performed, with the exception of "**$@**" as explained below under Pathname expansion is not performed. A is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the builtin command. The positional parameters are temporarily replaced when a shell function is executed (see below). When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces (see below). The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed. Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the special variable. That is, ''**$***'' is equivalent to ''**$1***c***$2***c***...**'', where is the first character of the value of the variable. If is null or unset, the parameters are separated by spaces. Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands as a separate word. That is, '' is equivalent to ''**$1**'' ''**$2**'' ... When there are no positional parameters, ''**$@**'' and expand to nothing (i.e. they are removed). Expands to the number of positional parameters in decimal. Expands to the status of the most recently executed foreground pipeline. Expands to the current option flags as specified upon invocation, by the builtin command, or those set by the shell itself (such as the flag). Expands to the process ID of the shell. In a () subshell, it expands to the process ID of the current shell, not the subshell. Expands to the process ID of the most recently executed background (asynchronous) command. Expands to the name of the shell or shell script. This is set at shell initialization. If is invoked with a file of commands, is set to the name of that file. Otherwise, it is set to the pathname used to invoke as given by argument zero. Expands to the last argument to the previous command, after expansion. Also set to the full pathname of each command executed and placed in the environment exported to that command. The following variables are set by the shell: The process ID of the shell's parent. The current working directory as set by the command. The previous working directory as set by the command. Set to the line of input read by the builtin command when no arguments are supplied. Expands to the user ID of the current user. Expands to the effective user ID of the current user. Expands to the full pathname used to invoke this instance of Expands to the version number of this instance of Incremented by one each time an instance of is started. Each time this parameter is referenced, a random integer is generated. The sequence of random numbers may be initialized by assigning a value to If is unset, it loses its special properties, even if it is subsequently reset. Each time this parameter is referenced, the number of seconds since shell invocation is returned. If a value is assigned to the value returned upon subsequent references is the number of seconds since the assignment plus the value assigned. If is unset, it loses its special properties, even if it is subsequently reset. Each time this parameter is referenced, the shell substitutes a decimal number representing the current sequential line number (starting with 1) within a script or function. When not in a script or function, the value substituted is not guaranteed to be meaningful. When in a function, the value is not the number of the source line that the command appears on (that information has been lost by the time the function is executed), but is an approximation of the number of executed in the current function. If is unset, it loses its special properties, even if it is subsequently reset. The value of the last option argument processed by the builtin command (see below). The index of the last option processed by the builtin command (see below). The following variables are used by the shell. In some cases, assigns a default value to a variable; these cases are noted below. The that is used for word splitting after expansion and to split lines into words with the builtin command. The default value is ''<space><tab><newline>''. The search path for commands. It is a colon-separated list of directories in which the shell looks for commands (see below). The default path is system–dependent, and is set by the administrator who installs A common value is ''.:/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:/etc:/usr/etc''. Note that in some circumstances, however, a leading '.' in can be a security hazard. The home directory of the current user; the default argument for the **cd** builtin command. The search path for the command. This is a colon-separated list of directories in which the shell looks for destination directories specified by the command. A sample value is ‘‘.:˜:/usr’’. If this parameter is set when **bash** is executing a shell script, its value is interpreted as a filename containing commands to initialize the shell, as in The value of is subjected to parameter expansion, command substitution, and arithmetic expansion before being interpreted as a pathname. is not used to search for the resultant pathname. If this parameter is set to a filename and the variable is not set, informs the user of the arrival of mail in the specified file. Specifies how often (in seconds) checks for mail. The default is 60 seconds. When it is time to check for mail, the shell does so before prompting. If this variable is unset, the shell disables mail checking. A colon-separated list of pathnames to be checked for mail. The message to be printed may be specified by separating the pathname from the message with a ‘?’. $_ stands for the name of the current mailfile. Example: **MAILPATH**=’/usr/spool/mail/bfox?"You have mail":˜/shell-mail?"$_ has mail!"’ supplies a default value for this variable, but the location of the user mail files that it uses is system dependent (e.g. /usr/spool/mail/**$USER**). If set, and a file that **bash** is checking for mail has been accessed since the last time it was checked, the message ‘‘The mail in‘ *mailfile* has been read’’ is printed. The value of this parameter is expanded (see below) and used as the primary prompt string. The default value is **‘‘bash\$ ’’**. The value of this parameter is expanded like and used as the secondary prompt string. The default is **‘‘bash>** **’’**. The value of this parameter is expanded like and the value is printed before each command displays during an execution trace. The first character of is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is ‘‘+ ’’. If set, the decoded prompt string does not undergo further expansion (see below). The number of commands to remember in the command history (see below). The name of the file in which command history is saved. (See below.) The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than that number of lines. If set to the value 1, displays error messages generated by the builtin command (see below). is initialized to 1 each time the shell is invoked or a shell script is executed. If set, the value is executed as a command prior to issuing each primary prompt. Controls the action of the shell on receipt of an character as the sole input. If set, the value is the number of consecutive characters typed before exits. If the variable exists but does not have a numeric value, or has no value, the default value is 10. If it does not exist, signifies the end of input to the shell. This is only in effect for interactive shells. Automatically set to a string that uniquely describes the type of machine on which is executing. The default is system-dependent. If set to a value greater than zero, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt. terminates after waiting for that number of seconds if input does not arrive. The default editor for the builtin command. A colon-separated list of suffixes to ignore when performing filename completion (see below). A filename whose suffix matches one of the entries in is excluded from the list of matched filenames. A sample value is ‘‘.o:˜’’. If set, reports terminated background jobs immediately, rather than waiting until before printing the next primary prompt. If set to a value of it means don’t enter lines which begin with a on the history list. If set to a value of it means don’t enter lines which match the last entered line. If unset, or if set to any other value than those above, all lines read by the parser are saved on the history list. If set, includes filenames beginning with a ‘.’ in the results of pathname expansion. If set, allows pathname patterns which match no files (see below) to expand to a null string, rather than themselves. The two characters which control history expansion and tokenization. The first character is the that is, the character which signals the start of a history expansion, normally ‘**!**’. The second character is the character which signifies that the remainder of the line is a comment, when found as the first character of a word. If set, the shell does not follow symbolic links when executing commands that change the current working directory. It uses the physical directory structure instead. By default, follows the logical chain of directories when performing commands such as 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 adds the contents of the new file to the already existing database. If set, does not overwrite an existing file with the and redirection operators. This variable may be overridden when creating output files by using the redirection operator instead of (see also the **–C** option to the builtin command). This variable controls how the shell interacts with the user and job control. If this variable is set, single word simple commands without redirections are treated as candidates for resumption of an existing stopped job. There is no ambiguity allowed; if there is more than one job beginning with the string typed, the job most recently accessed is selected. If this variable exists, the shell does not exit if it cannot execute the file specified in the command. If this is set, an argument to the builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to. If set, the and builtin commands do not print the current directory stack after successful execution. Expansion is performed on the command line after it has been split into words. There are seven kinds of expansion performed: and Only brace expansion, word splitting, and pathname expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The single exception to this is the expansion of ‘‘**$@**’’ as explained above (see is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *pathname expansion*, but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional followed by a series of comma-separated strings between a pair of braces, followed by an optional The preamble is prepended to each string contained within the braces, and the postamble is then appended to each resulting string, expanding left to right. Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example, a{d,c,b}e expands into ‘ade ace abe’. Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example: mkdir /usr/local/src/bash/{old,new,dist,bugs} or chown root

/usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}} Brace expansion introduces a slight incompatibility with traditional versions of the Bourne shell. does not treat opening or closing braces specially when they appear as part of a word, and preserves them in the output. removes braces from words as a consequence of brace expansion. For example, a word entered to as *file*{*1*,*2*} appears identically in the output. The same word is output as after expansion by If strict compatibility with is desired, start with the flag (see above) or disable brace expansion with the option to the command (see below). If a word begins with a tilde character (‘˜’), all of the characters preceding the first slash (or all characters, if there is no slash) are treated as a possible *login name*. If this *login name* is the null string, the tilde is replaced with the value of the parameter If is unset, the home directory of the user executing the shell is substituted instead. If a ‘+’ follows the tilde, the value of is substituted. If a ‘–’ follows, the value of is used. Additionally, each word is checked for unquoted instances of tildes following a or In these cases, tilde substitution is also performed. Consequently, one may use pathnames with tildes in and The ‘**$**’ character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name. ${*parameter*} The value of *parameter* is substituted. The braces are required when is a positional parameter with more than one digit, or when is followed by a character which is not to be interpreted as part of its name. In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. **Bash** tests for a parameter that is unset or null; omitting the colon results in a test only for a parameter that is unset. ${*parameter***:–***word*} **Use Default Values**. If is unset or null, the expansion of is substituted. Otherwise, the value of is substituted. ${*parameter***:=***word*} **Assign Default Values**. If is unset or null, the expansion of is assigned to The value of is then substituted. Positional parameters and special parameters may not be assigned to in this way. ${*parameter***:?***word*} **Display Error if Null or Unset**. If is null or unset, the expansion of *word* (or a message to that effect if is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of *parameter* is substituted. ${*parameter***:+***word*} **Use Alternate Value**. If is null or unset, nothing is substituted, otherwise the expansion of is substituted. ${#*parameter*} The length in characters of the value of *parameter* is substituted. If *parameter* is or the length substituted is the length of expanded within double quotes. ${*parameter*#*word*} ${*parameter*##*word*} The is expanded to produce a pattern just as in pathname expansion. If the pattern matches the beginning of the value of then the expansion is the value of with the shortest matching pattern deleted (the ‘‘#’’ case) or the longest matching pattern deleted (the ‘‘##’’ case). ${*parameter***%***word*} ${*parameter***%%***word*} The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches a trailing portion of the value of then the expansion is the value of with the shortest matching pattern deleted (the ‘‘**%**’’ case) or the longest matching pattern deleted (the ‘‘**%%**’’ case). *Command substitution* allows the output of a command to replace the command name. There are two forms: **$(***command***)** or **‘***command***‘** performs the expansion by executing *command* and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. When the old–style backquote form of substitution is used, backslash retains its literal meaning except when followed by or When using the $(*command*) form, all characters between the parentheses make up the command; none are treated specially. Command substitutions may be nested. To nest when using the old form, escape the inner backquotes with backslashes. If the substitution appears within double quotes, word splitting and pathname expansion are not performed on the results. Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is: **$[***expression***]** The is treated as if it were within double quotes, but a double quote inside the braces is not treated specially. All tokens in the expression undergo parameter expansion, command substitution, and quote removal. Arithmetic substitutions may be nested. The evaluation is performed according to the rules listed below under If is invalid, prints a message indicating failure and no substitution occurs. The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for The shell treats each character of as a delimiter, and splits the results of the other expansions into words on these characters. If the value of is exactly the default, then any sequence of characters serves to delimit words; otherwise each occurrence of an character is treated as a delimiter. If the value of is null, no word splitting occurs. cannot be unset. Explicit null arguments ("" or **’’**) are retained. Implicit null arguments, resulting from the expansion of that have no values, are removed. Note that if no expansion occurs, no splitting is performed. After word splitting, scans each for the characters and unless the flag has been set. If one of these characters appears, then the word is regarded as a and replaced with an alphabetically sorted list of pathnames matching the pattern. If no matching pathnames are found, and the shell variable is unset, the word is left unchanged. If the variable is set, the word is removed if no matches are found. When a pattern is used for pathname generation, the character at the start of a name or immediately following a slash must be matched explicitly, unless the shell variable is set. The slash character must always be matched explicitly. In other cases, the character is not treated specially. The special pattern characters have the following meanings: Matches any string, including the null string. Matches any single character. Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a any character lexically between those two characters, inclusive, is matched. If the first character following the is a or a then any character not enclosed is matched. A or may be matched by including it as the first or last character in the set. After the preceding expansions, all unquoted occurrences of the characters and " are removed. Before a command is executed, its input and output may be using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment. The following redirection operators may appear anywhere in a or may precede or follow a Redirections are processed in the order they appear, from left to right. In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is the redirection refers to the standard output (file descriptor 1). The word that follows the redirection operator in the following descriptions is subjected to brace expansion, tilde expansion, parameter expansion, command substitution, arithmetic expansion, quote removal, and pathname expansion. If it expands to more than one word, reports an error. Redirection of input causes the file whose name results from the expansion of to be opened for reading on file descriptor or the standard input (file descriptor 0) if is not specified. The general format for redirecting input is: [*n*]<*word* Redirection of output causes the file whose name results from the expansion of to be opened for writing on file descriptor or the standard output (file descriptor 1) if is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size. The general format for redirecting output is: [*n*]>*word* If the redirection operator is then the variable is not consulted, and the file is created regardless of the value of (see above). Redirection of output in this fashion causes the file whose name results from the expansion of to be opened for appending on file descriptor or the standard output (file descriptor 1) if is not specified. If the file does not exist it is created. The general format for appending output is: [*n*]>>*word* allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of with this construct. There are two formats for redirecting standard output and standard error: **&**>*word* and >**&***word* Of the two forms, the first is preferred. This is semantically equivalent to >*word* 2>**&**1 This type of redirection instructs the shell to read input from the current source until a line containing only (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input for a command. The format of here-documents is as follows:

<<[–]*word*
        here-document
delimiter

No parameter expansion, command substitution, pathname expansion, or arithmetic expansion is performed on If any characters in are quoted, the is the result of quote removal on and the lines in the here-document are not expanded. Otherwise, all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion. In the latter case, the pair is ignored, and must be used to quote the characters and If the redirection operator is then all leading tab characters are stripped from input lines and the line containing This allows here-documents within shell scripts to be indented in a natural fashion. The redirection operator [*n*]<**&***word* is used to duplicate input file descriptors. If expands to one or more digits, the file descriptor denoted by is made to be a copy of that file descriptor. If evaluates to file descriptor is closed. If is not specified, the standard input (file descriptor 0) is used. The operator [*n*]>**&***word* is used similarly to duplicate output file descriptors. If is not specified, the standard output (file descriptor 1) is used. The redirection operator [*n*]<>*word* causes the file whose name is the expansion of to be opened for both reading and writing on file descriptor or as the standard input and standard output if is not specified. Note that the order of redirections is significant. For example, the command ls > dirlist 2>**&**1 directs both standard output and standard error to the file while the command ls 2>**&**1 > dirlist directs only the standard output to file because the standard error was duplicated as standard output before the standard output was redirected to A shell function, defined as described above under stores a series of commands for later execution. However, functions are executed in the context of the current shell; no new process is created to interpret them (contrast this with the execution of a shell script). When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter is updated to reflect the change. Positional parameter 0 is unchanged. Variables local to the function may be declared with the builtin command. Ordinarily, variables and their values are shared between the function and its caller. If the builtin command is executed in a function, the function completes and execution resumes with the next command after the function call. When a function completes, the values of the positional parameters and the special parameter are restored to the values they had prior to function execution. Function names may be listed with the option to the or builtin commands. Functions may be exported so that subshells automatically have them defined with the option to the builtin. Functions may be recursive. No limit is imposed on the number of recursive calls. The shell maintains a list of that may be set and unset with the and builtin commands. The first word of each command is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The name and the replacement text may contain any valid shell input, including the listed above. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias to for instance, and does not try to recursively expand the replacement text. If the last character of the alias value is a then the next command word following the alias is also checked for alias expansion. Aliases are created and listed with the command, and removed with the command. There is no mechanism for using arguments in the replacement text, a la If arguments are needed, a shell function should be used. The rules concerning the definition and use of aliases are somewhat confusing. always reads at least one complete line of input before executing any of the commands on that line. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the next line of input is read. This means that the commands following the alias definition on that line are not affected by the new alias. This behavior is also an issue when functions are executed. Aliases are expanded when the function definition is read, not when the function is executed, because a function definition is itself a compound command. As a consequence, aliases defined in a function are not available until after that function is executed. To be safe, always put alias definitions on a separate line, and do not use in compound commands. Aliases are not expanded when the shell is not interactive. Note that for almost every purpose, aliases are superseded by shell functions. refers to the ability to selectively stop (*suspend*) the execution of processes and continue (*resume*) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the system's terminal driver and The shell associates a with each pipeline. It keeps a table of currently executing jobs, which may be listed with the command. When starts a job asynchronously (in the it prints a line that looks like: [1] 25647 indicating that this job is job number 1 and that the process ID of the single process in the job is 25647. uses the abstraction as the basis for job control. To facilitate the implementation of the user interface to job control, the system maintains the notion of a *current terminal process group ID*. Members of this process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as These processes are said to be in the processes are those whose process group ID differs from the terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or write to the terminal. Background processes which attempt to read from (write to) the terminal are sent a signal by the terminal driver, which, unless caught, causes the process to stop. If the operating system on which is running supports job control, allows you to use it. Typing the character (typically Control-Z) while a process is running causes that process to be stopped and returns you to Typing the character (typically Control-Y) causes the process to be stopped when it attempts to read input from the terminal, and control to be returned to You may then manipulate the state of this job, using the command to continue it in the background, the command to continue it in the foreground, or the command to kill it. A **^Z** takes effect immediately, and has the additional side effect of causing pending output and typeahead to be discarded. There are a number of ways to refer to a job in the shell. The character introduces a job name. Job number may be referred to as A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, refers to a stopped job. If a prefix matches more than one job, reports an error. Using on the other hand, would refer to any job containing the string in its command line. If the substring matches more than one job, reports an error. The symbols and refer to the shell's notion of the which is the last job stopped while it was in the foreground. The may be referenced using In output pertaining to jobs (e.g. the output of the command), the current job is always flagged with a and the previous job with a Simply naming a job can be used to bring it into the foreground: is a synonym for **''fg %1''**, bringing job 1 from the background into the foreground. Similarly, resumes job 1 in the background, equivalent to **''bg %1''**. The shell learns immediately whenever a job changes state. Normally, waits until it is about to print a prompt before reporting changes in a job's status so as to not interrupt any other output. If the variable is set, reports such changes immediately. (See also the option to the builtin command.) If you attempt to exit while jobs are stopped, the shell prints a message warning you. You may then use the command to inspect their status. If you do this, or try to exit again immediately, you are not warned again, and the stopped jobs are terminated. When **bash** is interactive, it ignores (so that **kill 0** does not kill an interactive shell), and is caught and handled (so that **wait** is interruptible). In all cases, **bash** ignores If job control is in effect, ignores and Synchronous jobs started by **bash** have signals set to the values inherited by the shell from its parent. Background jobs (jobs started with ignore and Commands run as a result of command substitution ignore the keyboard-generated job control signals and After a command has been split into words, if it results in a simple command and an optional list of arguments, the following actions are taken. If the command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, that function is invoked as described above in If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked. If the name is neither a shell function nor a builtin, and contains no slashes, searches each element of the for a directory containing an executable file by that name. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status. If the search is successful, or if the command name contains one or more slashes, the shell executes the named program. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments given, if any. If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a *shell script*, a file containing shell commands. A subshell is spawned to execute it. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to handle the script, with the exception that the locations of commands remembered by the parent (see below under **SHELL BUILTIN COMMANDS**) are retained by the child. If the program is a file beginning with the remainder of the first line specifies an interpreter for the program. The shell executes the specified interpreter on operating systems that do not handle this executable format themselves. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the program, followed by the name of the program, followed by the command arguments, if any. When a program is invoked it is given an array of strings called the This is a list of *name–value* pairs, of the form The shell allows you to manipulate the environment in several ways. On invocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for to child processes. Executed commands inherit the environment. The and commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the environment is modified, the new value becomes part of the environment, replacing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modified in the shell, less any pairs removed by the command, plus any additions via the and commands. The environment for any or function may be augmented temporarily by prefixing it with parameter assignments, as described above in These assignment statements affect only the environment seen by that command. If the flag is set (see the builtin command below), then parameter assignments are placed in the environment for a command, not just those that precede the command name. For the purposes of the shell, a command which exits with a zero exit status has succeeded. An exit status of zero indicates success. A non-zero exit status indicates failure. When a command terminates on a fatal signal, **bash** uses the value of 128+**signal** as the exit status. **Bash** itself returns the exit status of the last command executed, unless a syntax error occurs, in which case it exits with a non-zero value. See also the **exit** builtin command below. When executing interactively, displays the primary prompt when it is ready to read a command, and the secondary prompt when it needs more input to complete a command. allows the prompt to be customized by inserting a number of backslash-escaped special characters that are decoded as follows: the time the date CRLF the name of the shell, the basename of (the portion following the final slash) the current working directory the basename of the current working directory the username of the current user the hostname the command number of this command the history number of this command if the effective UID is 0, a otherwise a character code in octal a backslash After the string is decoded, if the variable is not set, it is expanded via parameter expansion, command substitution, arithmetic expansion, and word splitting. This is the library that handles reading input when using an interactive shell, unless the option is given. By default, the line editing commands are similar to those of emacs. A vi-style line editing interface is also available. In this section, the emacs-style notation is used to denote keystrokes. Control keys are denoted by C–*key*, e.g. C–n means Control–N. Similarly, keys are denoted by M–*key*, so M–x means Meta–X. (On keyboards without a key, M–*x* means ESC *x*, i.e. press the Escape key then the key. The combination M–C–*x* means ESC–Control–*x*, or press the Escape key then hold the Control key while pressing the key.) The default key-bindings may be changed with an *~/.inputrc* file. Other programs that use this library may add their own commands and bindings. For example, placing M–Control–u: universal-argument or C–Meta–u: universal-argument into the *~/.inputrc* would make M–C–u execute the command The following symbolic character names are recognized: RUBOUT, DEL, ESC, NEWLINE, SPACE, RETURN, LFD, TAB. Placing set editing-mode vi into a *~/.inputrc* file causes to start with a *vi*–like editing mode. The editing mode may be switched during interactive use by using the option to the builtin command (see below). You can have readline use a single line for display, scrolling the input between the two borders by placing set horizontal–scroll–mode On into a *~/.inputrc* file. The following is a list of the names of the commands and the default key-strokes to get them. Move to the start of the current line. Move to the end of the line. Move forward a character. Move back a character. Move forward to the end of the next word. Move back to the start of this, or the previous, word. Clear the screen leaving the current line at the top of the screen. Accept the line regardless of where the cursor is. If this line is non–empty, add it to the history list according to the state of the variable. If this line was a history line, then restore the history line to its original state. Fetch the previous command from the history list, moving back in the list. Fetch the next command from the history list, moving forward in the list. Move to the first line in the history, the first line entered. Move to the end of the input history, i.e., the line you are entering. Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search. Search forward starting at the current line and moving 'down' through the history as necessary. Expand the line the way the shell does when it reads it. This performs alias and history expansion. See below. Insert the last argument to the previous command (the last word on the previous line). Accept the current line for execution and fetch the next line relative to the current line from the history file for editing. 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 then return Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them. Add the next character that you type to the line verbatim. This is how to insert characters like C–q, for example. Insert a tab character. Insert the character typed. Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative arguments don't work. Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well. Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point. Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move point. Capitalize the current (or following) word. With a negative argument, do the previous word, but do not move point. Kill the text from the current cursor position to the end of the line. This saves the killed text on the kill–ring. (see below) Kill backward to the beginning of the line. This is normally unbound, in favor of **unix-line-discard**, which emulates the behavior of the standard Unix terminal driver. Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Kill the word behind the cursor. Do what C–u used to do in Unix line input. We save the killed text on the kill–ring, though. 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 the top of the kill ring into the buffer at point. Rotate the kill–ring, and yank the new top. Only works following 'yank' or 'yank–pop'. Add this digit to the argument already accumulating, or start a new argument. M–– starts a negative argument. Do what C–u does in By default, this is not bound to a key. Attempt to perform completion on the text before point. attempts completion treating the text as a variable (if the text begins with **$**), username (if the text begins with ~), hostname (if the text begins with @), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted. List the possible completions of the text before point. Attempt filename completion on the text before point. List the possible completions of the text before point, treating it as a filename. Attempt completion on the text before point, treating it as a username. List the possible completions of the text before point, treating it as a username. Attempt completion on the text before point, treating it as a shell variable. List the possible completions of the text before point, treating it as a shell variable. Attempt completion on the text before point, treating it as a hostname. List the possible completions of the text before point, treating it as a hostname. Abort the current editing command and ring the terminal's bell. Run the command that is bound to the uppercased key. Metafy the next character typed. This is for people without a meta key. f is equivalent to Incremental undo, separately remembered for each line. Undo all changes made to this line. This is like typing the 'undo' command enough times to get back to the beginning. Display version information about the current instance of When in editing mode, this causes a switch to editing mode. When in editing mode, this causes a switch to editing mode. The shell supports a history expansion feature that is similar to the history expansion in This section describes what syntax features are available. History expansion is performed immediately after a complete line is read, before the shell breaks it into words. It takes place in two parts. The first is determining which line from the previous history to use 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 the *event*, and the portions of that line that are acted upon are *words*. The line is broken into words in the same fashion as when reading input, so that several English, or Unix, words surrounded by quotes are considered as one word. Only backslash (\) can quote the history escape character, which is **!** by default. An event designator is a reference to a command line entry in the history list. Start a history substitution, except when followed by a <space>, <tab>, <newline>, = or (. Refer to the previous command. This is a synonym for '!–1'. Refer to command line Refer to the current command line minus Refer to the most recent command starting with Refer to the most recent command containing 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). 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. The zeroth word. For the shell, this is the command word. The *n*th word. The first argument. That is, word 1. The last argument. The word matched by the most recent '?*string*?' search. A range of words; '–*y*' abbreviates '0–*y*'. All of the words but the zeroth. 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. After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a ':'. Remove a trailing pathname component, leaving only the head. Remove a trailing suffix of the form ".xxx", leaving the basename. Remove all but the suffix. Remove all leading pathname components, leaving the tail. Print the new command but do not execute it. This takes effect immediately, so it should be the last specifier on the line. The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the **let** builtin command and **Arithmetic Expansion**). Evaluation is done in long integers with no check for overflow, though division by 0 is trapped and flagged as an error. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence. unary minus logical NOT multiplication, division, remainder addition, subtraction comparison equality and inequality assignment Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. The value of a parameter is coerced to a long integer within an expression. A shell variable need not have its integer attribute turned on to be used in an expression. Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above. **:** [*arguments*] No effect; the command does nothing beyond expanding and performing any specified redirections. A zero exit code is returned. **.** *filename* **source** *filename* Read and execute commands from in the current shell environment and return the exit status of the last command executed from Pathnames in are used to find the directory containing if does not contain a slash. The file searched for in need not be executable. The current directory is searched if no file is found in The return status is the status of the last command exited within the script (**true** if no commands are executed), and false if is not found. **alias** [*name*[=*value*] ...] **Alias** with no arguments prints the list of aliases in the form *name*=*value* on standard output. When arguments are supplied, an alias is defined for each *name* whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution when the alias is expanded. **Alias** returns true unless a *name* is given for which no alias has been defined. **bg** [*jobspec*] Place *jobspec* in the background, as if it had been started with If *jobspec* is not present, the shell's notion of the *current job* is used. **bind** [**–lvd**] [**-q** *name*] **bind -f** *filename* **bind** *keyseq*:*function-name* Display current key and function bindings, or bind a key sequence to a function or macro. The binding syntax accepted is identical to that of but each binding must be passed as a separate argument; e.g. '"\C-x\C-r": re-read-init-file'. Options, if supplied, have the following meanings: List the names of all **readline** functions List current function names and bindings Dump function names and bindings in such a way that they can be re-read Read key bindings from *filename* Query about which keys invoke the named *function* **break** [*n*] Exit from within a or loop. If *n* is specified, break *n* levels. must be ≥ 1. If is greater than the number of enclosing loops, all enclosing loops are exited. The return value is 0 unless the shell is not executing a loop when is executed. **builtin** [*shell–builtin* [*arguments*]] Execute the specified shell builtin, passing it and return its exit status. This is useful when you wish to define a function whose name is the same as a shell builtin, but need the functionality of the builtin within the function itself. The **cd** builtin is commonly redefined this way. **cd** [*dir*] Change the current directory to *dir*. The variable is the default The variable defines the search path for the directory containing Alternative directory names are separated by a colon (:). A null directory name in is the same as the current directory, i.e. ''**.**''. If begins with a slash (/), then is not used. An argument of is equivalent to The return value is true if the directory was successfully changed; false otherwise. **command** [**-p**] [*command* [*arg* ...]] Run with suppressing the normal shell function lookup. Only builtin commands or commands found in the are executed. If the option is given, the search for is performed using a default value for that is guaranteed to find all of the standard utilities. An argument of disables option checking for the rest of the arguments. If an error occurred or cannot be found, the exit status is 127. Otherwise, the exit status of the builtin is the exit status of **continue** [*n*] Resume the next iteration of the enclosing or loop. If is specified, resume at the *n*th enclosing loop. must be ≥ 1. If is greater than the number of enclosing loops, the last enclosing loop (the 'top–level' loop) is resumed. The return value is 0 unless the shell is not executing a loop when is executed.

declare [**–frxi**] [*name*[=*value*]] **typeset** [**–frxi**] [*name*[=*value*]] Declare variables and/or give them attributes. If no *name*s are given, then display the values of variables instead. Use function names only Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements. Mark *name*s for export to subsequent commands via the environment. The variable is treated as an integer; arithmetic evaluation (see is performed when the variable is assigned a value. Using '+' instead of '–' turns off the attribute instead. When used in a function, makes *name*s local, as with the command. Display the list of currently remembered directories. Directories are added to the list with the command; the command moves back up through the list. **echo** [**–ne**] [*arg* ...] Output the *arg*s, separated by spaces. If **–n** is specified, the trailing newline is suppressed. If the **–e** option is given, interpretation of the following backslash-escaped characters is enabled: alert (bell) backspace suppress trailing newline form feed new line carriage return horizontal tab vertical tab backslash the character whose ASCII code is *nnn* (octal) **enable** [**–n**] [*name* ...] Enable and disable builtin shell commands. This allows the execution of a disk command which has the same name as a shell builtin without specifying a full pathname. If **–n** is used, each *name* is disabled; otherwise, *names* are enabled. For example, to use the found in instead of the shell builtin version, type ''enable -n test''. **eval** [*arg* ...] The *arg*s are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of the command. If there are no or only null arguments, returns true. **exec** [[**–**] *command* [*arguments*]] If is specified, it replaces the shell. No new process is created. The become the arguments to *command*. If the first argument is the shell places a dash in the zeroth arg passed to This is what login does. If the file cannot be executed for some reason, the shell exits, unless the shell variable **no_exit_on_failed_exec** exists. If is not specified, any redirections take effect in the current shell. **exit** [*n*] **bye** [*n*] Cause the shell to exit with a status of *n*. If is omitted, the exit status is that of the last command executed. A trap on is executed before the shell terminates. **export** [**–npf**] [*name*[=*word*]] ... The supplied are marked for automatic export to the environment of subsequently executed commands. If the option is given, the refer to functions. If no are given, or if the option is supplied, a list of all names that are exported in this shell is printed. The option causes the export property to be removed from the named variables. An argument of disables option checking for the rest of the arguments. returns an exit status of true unless an illegal option is encountered. **fc** [**–e** *ename*] [**–nlr**] [*first*] [*last*] **fc –s** [*pat*=*rep*] [*cmd*] Fix Command. In the first form, a range of commands from to is selected from the history list. and may be specified as a string (to locate the last 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 is not specified it is set to the current command for listing (so that prints the last 10 commands) and to otherwise. If is not specified it is set to the previous command for editing and –16 for listing.

The flag suppresses the command numbers when listing. The flag reverses the order of the commands. If the flag is given, the commands are listed on standard output. Otherwise, the editor given by is invoked on a file containing those commands. If is not given, the value of the variable is used, and the value of if is not set. If neither variable is set, *vi* is used. When editing is complete, the edited commands are echoed and executed.

In the second form, the command is re-executed after the substitution *old*=*new* is performed. A useful alias to use with this is ''r=fc –s'', so that typing ''r cc'' runs the last command beginning with ''cc'' and typing ''r'' re-executes the last command. **fg** [*jobspec*] Place in the foreground, and make it the current job. If is not present, the shell's notion of the *current job* is used. **getopts** *optstring name* [*args*] is used by shell procedures to parse positional parameters. contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. Each time it is invoked, places the next option in the shell variable initializing if it does not exist, and the index of the next argument to be processed into the variable is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, places that argument into the variable

can report errors in two ways. If the first character of is a colon, error reporting is used. In normal operation diagnostic messages are printed when illegal options or missing option arguments are encountered. If the variable is set to 0, no error message will be displayed, even if the first character of is not a colon.

If an illegal option is seen, places ? into and, if not silent, prints an error message and unsets If is silent, the option character found is placed in and no diagnostic message is printed.

If a required argument is not found, and is not silent, a question mark (**?**) is placed in is unset, and a diagnostic message is printed. If is silent, then a colon (**:**) is placed in and is set to the option character found.

normally parses the positional parameters, but if more arguments are given in parses those instead. returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs. **hash** [**–r**] [*name*] For each the full pathname of the command is determined and remembered. The option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is printed. An argument of disables option checking for the rest of the arguments. The return status is true unless a is not found or an illegal option is supplied. **help** [*pattern*] Display helpful information about builtin commands. If is specified, gives detailed help on all commands matching otherwise a list of the builtins is printed. **history** [*n*] **history –rwan** [*filename*] With no options, display the command history list with line numbers. Lines listed with with a have been modified. An argument of lists only the last lines. If a non-option argument is supplied, it is used as the name of the history file; if not, the value of (default ~/*.bash_history*) is used. Options, if supplied, have the following meanings: Append the ''new'' history lines (history lines entered since the beginning of the current bash session) to the history file Read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current bash session. read the contents of the history file and use them as the current history write the current history to the history file, overwriting the history file's contents. **jobs** [**–lnp**] [*jobspec* ...] Lists the active jobs. The option lists process IDs in addition to the normal information; the option lists only the process ID of the job's process group leader. The option displays only jobs that have changed status since last notfied. If is given, output is restricted to information about that job. **kill** [**-s sigspec** | **–sigspec**] [*pid* | *jobspec*] ... **kill –l** [*signum*] Send the signal named by to the processes named by or is either a signal name such as or a signal number. If is a signal name, the name is case insensitive and may be given with or without the prefix. If is not present, then is assumed. An argument of lists the signal names. If any arguments are supplied when is given, the names of the specified signals are listed. An argument of disables option checking for the rest of the arguments. returns true if at least one signal was successfully sent, or false if an error occurs. **let** *arg* [*arg* ...] Each is an arithmetic expression to be evaluated (see If the last evaluates to 0, returns 1; 0 is returned otherwise. **local** [*name*[=*value*]] Create a local variable named and assign it When is used within a function, it causes the variable to have a visible scope restricted to that function and its children. With no operands, writes a list of local variables to the standard output. It is an error to use when not within a function. Exit a login shell. **popd** [**+/–n**] Removes entries from the directory stack. With no arguments, removes the top directory from the stack, and performs a to the new top directory. removes the *n*th entry counting from the left of the list shown by starting with zero. For example: ''popd +0'' removes the first directory, ''popd +1'' the second. removes the *n*th entry counting from the right of the list shown by starting with zero. For example: ''popd -0'' removes the last directory, ''popd -1'' the next to last. If the variable is unset and the command is successful, a is performed as well. **pushd** *dir* **pushd +/–n** Adds a directory to the top of the directory stack, or rotates the stack, making the new top of the stack the current working directory. With no arguments, exchanges the top two directories. Rotates the stack so that the *n*th directory (counting from the left of the list shown by is at the top. Rotates the stack so that the *n*th directory (counting from the right) is at the top. adds to the directory stack at the top, making it the new current working directory. If the variable is not set and the command is successful, a is performed as well. **pwd** Print the absolute pathname of the current working directory. The path printed contains no symbolic links (but see the description of under above). **read** [**–r**] [*name* ...] One line is read from the standard input, and the first word is assigned to the first the second word to the second and so on, with leftover words assigned to the last Only the characters in are recognized as word delimiters. The return code is zero, unless end-of-file is encountered. If the option is given, a backslash-newline pair is not ignored, and the backslash is considered to be part of the line. **readonly** [**–pf**] [*name* ...] The given *names* are marked readonly and the values of these *names* may not be changed by subsequent assignment. If the option is supplied, the functions corresponding to the *names* are so marked. If no arguments are given, or if the option is supplied, a list of all readonly names is printed. An argument of disables option checking for the rest of the arguments. **return** [*n*] Causes a function to exit with the return value specified by If is omitted, the return status is that of the last command executed in the function body. If used outside a function, but during execution of a script by the (**source**) command, it causes the shell to stop executing that script and return either or the exit status of the last command executed within the script as the exit status of the script. **set** [**–aefhknotuvxldCH**] [*arg* ...] Automatically mark variables which are modified or created for export to the environment of subsequent commands. Exit immediately if a *simple-command* (see above) exits with a non-zero status. The shell does not exit if the command that fails is part of an or loop, part of an statement, part of a or list, or if the command's return value is being inverted via Disable pathname expansion. Locate and remember function commands as functions are defined. Function commands are normally looked up when the function is executed. All keyword arguments are placed in the environment for a command, not just those that precede the command name. Monitor mode. Job control is enabled. This flag is on by default for interactive shells on systems that support it (see above). Background processes run in a separate process group and a line containing their exit status is printed upon their completion. Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored for interactive shells. The *option-name* can be one of the following: Same as The shell performs curly brace expansion (see above). This is on by default. Use an emacs-style command line editing interface. Same as Same as The effect is as if the shell command 'IGNOREEOF=10' had been executed (see above). Same as Same as Same as Same as Same as The effect is as if the shell command 'notify=' had been executed (see above). Same as Same as Use a vi-style command line editing interface. Same as If no *option-name* is supplied, the values of the current options are printed. Exit after reading and executing one command. Treat unset variables as an error when performing parameter expansion. If expansion is attempted on an unset variable, the shell prints an error message, and, if not interactive, exits with a non-zero status. Print shell input lines as they are read. After expanding each displays the expanded value of followed by the command and its expanded arguments. Save and restore the binding of *name* in a **for** *name* [in **word**] command (see above). 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. The effect is as if the shell command 'noclobber=' had been executed (see above). Enable style history substitution. This flag is on by by default. If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the *arg*s, even if some of them begin with a Signal the end of options, cause all remaining *arg*s to be assigned to the positional parameters. The and options are turned off. If there are no *arg*s, the positional parameters remain unchanged. Using + rather than – causes these flags to be turned off. The flags can also be specified as options to an invocation of the shell. The current set of flags may be found in After the option arguments are processed, the remaining *arg*s are treated as values for the positional parameters and are assigned, in order, to If no options or *arg*s are supplied, all shell variables are printed. The return status is always true unless an illegal option is encountered. **shift** [*n*] The positional parameters from *n*+1 ... are renamed to If is not given, it is assumed to be 1. The exit status is 1 if is greater than otherwise 0. **suspend** [**–f**] Suspend the execution of this shell until it receives a signal. The option says not to complain if this is a login shell; just suspend anyway. **test** *expr* **[** *expr* **]** Return a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. True if *file* exists and is block special. True if *file* exists and is character special. True if *file* exists and is a directory. True if *file* exists True if *file* exists and is a regular file. True if *file* exists and is set-group-id. True if *file* has its ''sticky'' bit set. True if *file* exists and is a symbolic link. True if *file* exists and is a named pipe. True if *file* exists and is readable. True if *file* exists and has a size greater than zero. True if *file* exists and is a socket. True if is opened on a terminal. If is omitted, it defaults to 1 (standard output). True if *file* exists and its set-user-id bit is set. True if *file* exists and is writable. True if *file* exists and is executable. True if *file* exists and is owned by the effective user id. True if *file* exists and is owned by the effective group id. *file1* **–nt** *file2* True if *file1* is newer (according to modification date) than *file2*. *file1* **–ot** *file2* True if *file1* is older than file2. *file1* **–ef** *file* True if *file1* and *file2* have the same device and inode numbers. True if the length of *string* is zero. *string* True if the length of is non-zero. *string1* **=** *string2* True if the strings are equal. *string1* **!=** *string2* True if the strings are not equal. True if is false. *expr1* **–a** *expr2* True if both AND are true. *expr1* **–o** *expr2* True if either OR is true. is one of or These arithmetic binary operators return true if *arg1* is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than *arg2*, respectively. and may be positive integers, negative integers, or the special expression **–l** *string*, which evaluates to the length of Print the accumulated user and system times for the shell and for processes run from the shell. **trap** [*arg*] [*sigspec*] The command is to be read and executed when the shell receives signal(s) If is absent or all specified signals are are reset to their original values (the values they had upon entrance to the shell). If is the null string this signal is ignored by the shell and by the commands it invokes. is either a signal name in <signal.h>, or a signal number. If is (0) the command is executed on exit from the shell. With no arguments, prints the list of commands associated with each signal number. The option causes the shell to print a list of signal names and their corresponding numbers. An argument of disables option checking for the rest of the arguments. Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created. The return status is false if either then trap name or number is invalid; otherwise returns true. **type** [**–all**] [**–type** | **–path**] [*name* ...] With no options, indicate how each would be interpreted if used as a command name. If the flag is used, prints a phrase which is one of or if is an alias, shell reserved word, function, builtin, or disk file, respectively. If the name is not found, then nothing is printed, and an exit status of false is returned. If the flag is used, either returns the name of the disk file that would be executed if were specified as a command name, or nothing if would not return If a command is hashed, prints the hashed value, not necessarily the file that appears first in If the flag is used, prints all of the places that contain an executable named This includes aliases and functions, if and only if the flag is not also used. The table of hashed commands is not consulted when using accepts and in place of and respectively. An argument of disables option checking for the rest of the arguments. returns true if any of the arguments are found, false if none are found. **ulimit** [**–SHacdfmstpn** [*limit*]] provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The value of can be a number in the unit specified for the resource, or the value The **H** and **S** options specify that the hard or soft limit is set for the given resource. A hard limit cannot be increased once it is set; a soft limit may be increased up to the value of the hard limit. If neither **H** nor **S** is specified, the command applies to the soft limit. If is omitted, the current value of the soft limit of the resource is printed, unless the **H** option is given. When more than one resource is specified, the limit name and unit is printed before the value. Other options are interpreted as follows: all current limits are reported the maximum size of core files created the maximum size of a process's data segment the maximum size of files created by the shell the maximum resident set size the maximum stack size the maximum amount of cpu time in seconds the pipe size in 512-byte blocks (this may not be set) the maximum number of open file descriptors (most systems do not allow this value to be set, only displayed) An argument of disables option checking for the rest of the arguments. If is given, it is the new value of the specified resource (the option is display only). If no option is given, then is assumed. Values are in 1024-byte increments, except for which is in seconds, and which is in units of 512-byte blocks. **umask** [**–S**] [*mode*] The user file-creation mask is set to If begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by If is omitted, or if the option is supplied, the current value of the mask is printed. The option causes the mask to be printed in symbolic form; the default output is an octal number. An argument of disables option checking for the rest of the arguments. **unalias** [*name* ...] Remove *name*s from the list of defined aliases. The return value is true unless is not a defined alias. **unset** [**–fv**] [*name* ...] For each remove the corresponding variable or, given the option, function. An argument of disables option checking for the rest of the arguments. Note that and cannot be unset. If any of or are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless the variable does not exist or is non-unsettable. **wait** [*n*] Wait for the specified process and report its termination status. may be a process ID or a job specification; if a job spec is given, all processes in that job's pipeline are waited for. If is not given, all currently active child processes are waited for, and the return code is zero. A *login shell* is one whose first character of argument zero is a or one started with the flag. An *interactive* shell is one whose standard input and output are both connected to terminals (as determined by or one started with the flag. is set and includes if is interactive, allowing a way to test this state from a shell script or a startup file.

Login shells:
  On login:
        if /*etc*/*profile* exists, source it.

        if ˜/.*bash_profile* exists, source it,
          else if ˜/.*bash_login* exists, source it,
            else if ˜/.*profile* exists, source it.

  On logout:
        if ˜/.*bash_logout* exists, source it.

Non-login interactive shells:
  On startup:
        if ˜/.*bashrc* exists, source it.

Non-interactive shells:
  On startup:
        if the environment variable **ENV** is non-null, expand
          it and source the file it names.

The Gnu Readline Library, Brian Fox *The Gnu History Library*, Brian Fox *A System V Compatible Implementation of 4.2BSD Job Control*, David Lennert *How to wear weird pants for fun and profit*, Brian Fox *sh*(1), *ksh*(1), *csh*(1) /*bin*/*bash* The **bash** executable /*etc*/*profile* The systemwide initialization file, executed for login shells ˜/.*bash_profile* The personal initialization file, executed for login shells ˜/.*bashrc* The individual per-interactive-shell startup file ˜/.*inputrc* Individual *Readline* initialization file Brian Fox, Free Software Foundation (primary author)

bfox@ai.MIT.Edu Chet Ramey, Case Western Reserve University

chet@ins.CWRU.Edu If you find a bug in you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of that you have. Once you have determined that a bug actually exists, mail a bug report to *bash–maintainers@ai.MIT.Edu*. If you have a fix, you are welcome to mail that as well! Suggestions and ‘philosophical’ bug reports may be mailed to *bug-bash@ai.MIT.Edu* or posted to the Usenet newsgroup ALL bug reports should include: The version number of **bash** The hardware and operating system The compiler used to compile A description of the bug behaviour A short script or ‘recipe’ which exercises the bug Comments and bug reports concerning this manual page should be directed to It’s too big and too slow. There are some subtle differences between and traditional versions of mostly because of the specification. Aliases are confusing in some uses.