Bash – The GNU shell*

Chet Ramey
Case Western Reserve University
chet@po.cwru.edu

1. Introduction

Bash is the shell, or command language interpreter, that will appear in the GNU operating system. The name is an acronym for the "Bourne-Again SHell", a pun on Steve Bourne, the author of the direct ancestor of the current UNIX® shell */bin/sh*, which appeared in the Seventh Edition Bell Labs Research version of UNIX.

Bash is an **sh**–compatible shell that incorporates useful features from the Korn shell (**ksh**) and the C shell (**csh**), described later in this article. It is ultimately intended to be a conformant implementation of the IEEE POSIX Shell and Utilities specification (IEEE Working Group 1003.2). It offers functional improvements over sh for both interactive and programming use.

While the GNU operating system will most likely include a version of the Berkeley shell csh, Bash will be the default shell. Like other GNU software, Bash is quite portable. It currently runs on nearly every version of UNIX and a few other operating systems – an independently-supported port exists for OS/2, and there are rumors of ports to DOS and Windows NT. Ports to UNIX-like systems such as QNX and Minix are part of the distribution.

The original author of Bash was Brian Fox, an employee of the Free Software Foundation. The current developer and maintainer is Chet Ramey, a volunteer who works at Case Western Reserve University.

2. What's POSIX, anyway?

POSIX is a name originally coined by Richard Stallman for a family of open system standards based on UNIX. There are a number of aspects of UNIX under consideration for standardization, from the basic system services at the system call and C library level to applications and tools to system administration and management. Each area of standardization is assigned to a working group in the 1003 series.

The POSIX Shell and Utilities standard has been developed by IEEE Working Group 1003.2 (POSIX.2).‡ It concentrates on the command interpreter interface and utility programs commonly executed from the command line or by other programs. An initial version of the standard has been approved and published by the IEEE, and work is currently underway to update it. There are four primary areas of work in the 1003.2 standard:

- Aspects of the shell's syntax and command language. A number of special builtins such as **cd** and **exec** are being specified as part of the shell, since their functionality usually cannot be implemented by a separate executable;
- A set of utilities to be called by shell scripts and applications. Examples are programs like *sed, tr,* and *awk.* Utilities commonly implemented as shell builtins are described in this section, such as **test** and **kill**. An expansion of this section's scope, termed the User Portability Extension, or UPE, has standardized interactive programs such as *vi* and *mailx;*
- A group of functional interfaces to services provided by the shell, such as the traditional `system()` C library function. There are functions to perform shell word expansions, perform filename expansion (*globbing*), obtain values of POSIX.2 system configuration variables, retrieve values of environment variables (`getenv()`), and other services;

---

*An earlier version of this article appeared in The Linux Journal.

‡IEEE, *IEEE Standard for Information Technology -- Portable Operating System Interface (POSIX) Part 2: Shell and Utilities*, 1992.

- A suite of “development” utilities such as *c89* (the POSIX.2 version of *cc*), and *yacc.*

Bash is concerned with the aspects of the shell’s behavior defined by POSIX.2. The shell command language has of course been standardized, including the basic flow control and program execution constructs, I/O redirection and pipelining, argument handling, variable expansion, and quoting. The *special* builtins, which must be implemented as part of the shell to provide the desired functionality, are specified as being part of the shell; examples of these are **eval** and **export**. Other utilities appear in the sections of POSIX.2 not devoted to the shell which are commonly (and in some cases must be) implemented as builtin commands, such as **read** and **test**. POSIX.2 also specifies aspects of the shell’s interactive behavior as part of the UPE, including job control and command line editing. Interestingly enough, only *vi*-style line editing commands have been standardized; *emacs* editing commands were left out due to objections.

While POSIX.2 includes much of what the shell has traditionally provided, some important things have been omitted as being “beyond its scope.” There is, for instance, no mention of a difference between a *login* shell and any other interactive shell (since POSIX.2 does not specify a login program). No fixed startup files are defined, either – the standard does not mention *.profile*.

3. Basic Bash features

Since the Bourne shell provides Bash with most of its philosophical underpinnings, Bash inherits most of its features and functionality from sh. Bash implements all of the traditional sh flow control constructs (*for*, *if*, *while*, etc.). All of the Bourne shell builtins, including those not specified in the POSIX.2 standard, appear in Bash. Shell *functions*, introduced in the SVR2 version of the Bourne shell, are similar to shell scripts, but are defined using a special syntax and are executed in the same process as the calling shell. Bash has shell functions which behave in a fashion upward-compatible with sh functions. There are certain shell variables that Bash interprets in the same way as sh, such as **PS1**, **IFS**, and **PATH**. Bash implements essentially the same grammar, parameter and variable expansion semantics, redirection, and quoting as the Bourne shell. Where differences appear between the POSIX.2 standard and traditional sh behavior, Bash follows POSIX.

The Korn Shell (**ksh**) is a descendent of the Bourne shell written at AT&T Bell Laboratories by David Korn†. It provides a number of useful features that POSIX and Bash have adopted. Many of the interactive facilities in POSIX.2 have their roots in the ksh: for example, the POSIX and ksh job control facilities are nearly identical. Bash includes features from the Korn Shell for both interactive use and shell programming. For programming, Bash provides variables such as **RANDOM** and **REPLY**, the **typeset** builtin, the ability to remove substrings from variables based on patterns, and shell arithmetic. **RANDOM** expands to a random number each time it is referenced; assigning a value to **RANDOM** seeds the random number generator. **REPLY** is the default variable used by the **read** builtin when no variable names are supplied as arguments. The **typeset** builtin is used to define variables and give them attributes such as **readonly**. Bash arithmetic allows the evaluation of an expression and the substitution of the result. Shell variables may be used as operands, and the result of an expression may be assigned to a variable. Nearly all of the operators from the C language are available, with the same precedence rules:

```
$ echo $((3 + 5 * 32))
163
```

For interactive use, Bash implements ksh-style aliases and builtins such as **fc** (discussed below) and **jobs**. Bash aliases allow a string to be substituted for a command name. They can be used to create a mnemonic for a UNIX command name (`alias del=rm`), to expand a single word to a complex command (`alias news='xterm -g 80x45 -title trn -e trn -e -S1 -N &'`), or to ensure that a command is invoked with a basic set of options (`alias ls="/bin/ls -F"`).

The C shell (**csh**)†, originally written by Bill Joy while at Berkeley, is widely used and quite popular for its interactive facilities. Bash includes a csh-compatible history expansion mechanism (“! history”), brace expansion, access to a stack of directories via the **pushd**, **popd**, and **dirs** builtins, and tilde expansion, to generate users’ home directories. Tilde expansion has also been adopted by both the Korn Shell and

†Morris Bolsky and David Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.

POSIX.2.

There were certain areas in which POSIX.2 felt standardization was necessary, but no existing implementation provided the proper behavior. The working group invented and standardized functionality in these areas, which Bash implements. The **command** builtin was invented so that shell functions could be written to replace builtins; it makes the capabilities of the builtin available to the function. The reserved word "!" was added to negate the return value of a command or pipeline; it was nearly impossible to express "if not x" cleanly using the sh language. There exist multiple incompatible implementations of the **test** builtin, which tests files for type and other attributes and performs arithmetic and string comparisons. POSIX considered none of these correct, so the standard behavior was specified in terms of the number of arguments to the command. POSIX.2 dictates exactly what will happen when four or fewer arguments are given to **test**, and leaves the behavior undefined when more arguments are supplied. Bash uses the POSIX.2 algorithm, which was conceived by David Korn.

3.1. Features not in the Bourne Shell

There are a number of minor differences between Bash and the version of sh present on most other versions of UNIX. The majority of these are due to the POSIX standard, but some are the result of Bash adopting features from other shells. For instance, Bash includes the new "!" reserved word, the **command** builtin, the ability of the **read** builtin to correctly return a line ending with a backslash, symbolic arguments to the **umask** builtin, variable substring removal, a way to get the length of a variable, and the new algorithm for the **test** builtin from the POSIX.2 standard, none of which appear in sh.

Bash also implements the "$(...)" command substitution syntax, which supersedes the sh \`...\` construct. The "$(...)" construct expands to the output of the command contained within the parentheses, with trailing newlines removed. The sh syntax is accepted for backwards compatibility, but the "$(...)" form is preferred because its quoting rules are much simpler and it is easier to nest.

The Bourne shell does not provide such features as brace expansion, the ability to define a variable and a function with the same name, local variables in shell functions, the ability to enable and disable individual builtins or write a function to replace a builtin, or a means to export a shell function to a child process.

Bash has closed a long-standing shell security hole by not using the **$IFS** variable to split each word read by the shell, but splitting only the results of expansion (ksh and the 4.4 BSD sh have fixed this as well). Useful behavior such as a means to abort execution of a script read with the "." command using the **return** builtin or automatically exporting variables in the shell's environment to children is also not present in the Bourne shell. Bash provides a much more powerful environment for both interactive use and programming.

4. Bash-specific Features

This section details a few of the features which make Bash unique. Most of them provide improved interactive use, but a few programming improvements are present as well. Full descriptions of these features can be found in the Bash documentation.

4.1. Startup Files

Bash executes startup files differently than other shells. The Bash behavior is a compromise between the csh principle of startup files with fixed names executed for each shell and the sh "minimalist" behavior. An interactive instance of Bash started as a login shell reads and executes *~/.bash_profile* (the file .bash_profile in the user's home directory), if it exists. An interactive non-login shell reads and executes *~/.bashrc*. A non-interactive shell (one begun to execute a shell script, for example) reads no fixed startup file, but uses the value of the variable **$ENV**, if set, as the name of a startup file. The ksh practice of reading **$ENV** for every shell, with the accompanying difficulty of defining the proper variables and functions for interactive and non-interactive shells or having the file read only for interactive shells, was considered too complex. Ease of use won out here. Interestingly, the next release of ksh will change to reading **$ENV**

---

†Bill Joy, An Introduction to the C Shell, *UNIX User's Supplementary Documents*, University of California at Berkeley, 1986.

only for interactive shells.

4.2. New Builtin Commands

There are a few builtins which are new or have been extended in Bash. The **enable** builtin allows builtin commands to be turned on and off arbitrarily. To use the version of *echo* found in a user's search path rather than the Bash builtin, `enable -n echo` suffices. The **help** builtin provides quick synopses of the shell facilities without requiring access to a manual page. **Builtin** is similar to **command** in that it bypasses shell functions and directly executes builtin commands. Access to a csh-style stack of directories is provided via the **pushd**, **popd**, and **dirs** builtins. **Pushd** and **popd** insert and remove directories from the stack, respectively, and **dirs** lists the stack contents. On systems that allow fine-grained control of resources, the **ulimit** builtin can be used to tune these settings. **Ulimit** allows a user to control, among other things, whether core dumps are to be generated, how much memory the shell or a child process is allowed to allocate, and how large a file created by a child process can grow. The **suspend** command will stop the shell process when job control is active; most other shells do not allow themselves to be stopped like that. **Type,** the Bash answer to **which** and **whence,** shows what will happen when a word is typed as a command:

```
$ type export
export is a shell builtin
$ type -t export
builtin
$ type bash
bash is /bin/bash
$ type cd
cd is a function
cd ()
{
    builtin cd ${1+"$@"} && xtitle $HOST: $PWD
}
```

Various modes tell what a command word is (reserved word, alias, function, builtin, or file) or which version of a command will be executed based on a user's search path. Some of this functionality has been adopted by POSIX.2 and folded into the **command** utility.

4.3. Editing and Completion

One area in which Bash shines is command line editing. Bash uses the *readline* library to read and edit lines when interactive. Readline is a powerful and flexible input facility that a user can configure to individual tastes. It allows lines to be edited using either emacs or vi commands, where those commands are appropriate. The full capability of emacs is not present – there is no way to execute a named command with M-x, for instance – but the existing commands are more than adequate. The vi mode is compliant with the command line editing standardized by POSIX.2.

Readline is fully customizable. In addition to the basic commands and key bindings, the library allows users to define additional key bindings using a startup file. The *inputrc* file, which defaults to the file *~/.inputrc*, is read each time readline initializes, permitting users to maintain a consistent interface across a set of programs. Readline includes an extensible interface, so each program using the library can add its own bindable commands and program-specific key bindings. Bash uses this facility to add bindings that perform history expansion or shell word expansions on the current input line.

Readline interprets a number of variables which further tune its behavior. Variables exist to control whether or not eight-bit characters are directly read as input or converted to meta-prefixed key sequences (a meta-prefixed key sequence consists of the character with the eighth bit zeroed, preceded by the *meta-prefix* character, usually escape, which selects an alternate keymap), to decide whether to output characters with the eighth bit set directly or as a meta-prefixed key sequence, whether or not to wrap to a new screen line when a line being edited is longer than the screen width, the keymap to which subsequent key bindings should apply, or even what happens when readline wants to ring the terminal's bell. All of these variables can be set in the inputrc file.

The startup file understands a set of C preprocessor-like conditional constructs which allow variables or key bindings to be assigned based on the application using readline, the terminal currently being used, or the editing mode. Users can add program-specific bindings to make their lives easier: I have bindings that let me edit the value of **$PATH** and double-quote the current or previous word:

```
# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word -- insert open and close double
# quotes and move to just after the open quote
"\C-x\"": "\"\"\C-b"
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

There is a readline command to re-read the file, so users can edit the file, change some bindings, and begin to use them almost immediately.

Bash implements the **bind** builtin for more dyamic control of readline than the startup file permits. **Bind** is used in several ways. In *list* mode, it can display the current key bindings, list all the readline editing directives available for binding, list which keys invoke a given directive, or output the current set of key bindings in a format that can be incorporated directly into an inputrc file. In *batch* mode, it reads a series of key bindings directly from a file and passes them to readline. In its most common usage, **bind** takes a single string and passes it directly to readline, which interprets the line as if it had just been read from the inputrc file. Both key bindings and variable assignments may appear in the string given to **bind**.

The readline library also provides an interface for *word completion*. When the *completion* character (usually TAB) is typed, readline looks at the word currently being entered and computes the set of filenames of which the current word is a valid prefix. If there is only one possible completion, the rest of the characters are inserted directly, otherwise the common prefix of the set of filenames is added to the current word. A second TAB character entered immediately after a non-unique completion causes readline to list the possible completions; there is an option to have the list displayed immediately. Readline provides hooks so that applications can provide specific types of completion before the default filename completion is attempted. This is quite flexible, though it is not completely user-programmable. Bash, for example, can complete filenames, command names (including aliases, builtins, shell reserved words, shell functions, and executables found in the file system), shell variables, usernames, and hostnames. It uses a set of heuristics that, while not perfect, is generally quite good at determining what type of completion to attempt.

4.4. History

Access to the list of commands previously entered (the *command history*) is provided jointly by Bash and the readline library. Bash provides variables (**$HISTFILE**, **$HISTSIZE**, and **$HISTCONTROL**) and the **history** and **fc** builtins to manipulate the history list. The value of **$HISTFILE** specifes the file where Bash writes the command history on exit and reads it on startup. **$HISTSIZE** is used to limit the number of commands saved in the history. **$HISTCONTROL** provides a crude form of control over which commands are saved on the history list: a value of *ignorespace* means to not save commands which begin with a space; a value of *ignoredups* means to not save commands identical to the last command saved. **$HISTCONTROL** was named **$history_control** in earlier versions of Bash; the old name is still accepted for backwards compatibility. The **history** command can read or write files containing the history list and display the current list contents. The **fc** builtin, adopted from POSIX.2 and the Korn Shell, allows display and re-execution, with optional editing, of commands from the history list. The readline library offers a set of commands to search the history list for a portion of the current input line or a string typed by the user. Finally, the *history* library, generally incorporated directly into the readline library, implements a facility for history recall, expansion, and re-execution of previous commands very similar to csh (“bang history”, so called because the exclamation point introduces a history substitution):

```
$ echo a b c d e
a b c d e
$ !! f g h i
echo a b c d e f g h i
a b c d e f g h i
$ !-2
echo a b c d e
a b c d e
$ echo !-2:1-4
echo a b c d
a b c d
```

The command history is only saved when the shell is interactive, so it is not available for use by shell scripts.

4.5. New Shell Variables

There are a number of convenience variables that Bash interprets to make life easier. These include **FIGNORE**, which is a set of filename suffixes identifying files to exclude when completing filenames; **HOSTTYPE**, which is automatically set to a string describing the type of hardware on which Bash is currently executing; **command_oriented_history**, which directs Bash to save all lines of a multiple-line command such as a *while* or *for* loop in a single history entry, allowing easy re-editing; and **IGNOREEOF**, whose value indicates the number of consecutive EOF characters that an interactive shell will read before exiting – an easy way to keep yourself from being logged out accidentally. The **auto_resume** variable alters the way the shell treats simple command names: if job control is active, and this variable is set, single-word simple commands without redirections cause the shell to first look for and restart a suspended job with that name before starting a new process.

4.6. Brace Expansion

Since sh offers no convenient way to generate arbitrary strings that share a common prefix or suffix (filename expansion requires that the filenames exist), Bash implements *brace expansion*, a capability picked up from csh. Brace expansion is similar to filename expansion, but the strings generated need not correspond to existing files. A brace expression consists of an optional *preamble*, followed by a pair of braces enclosing a series of comma-separated strings, and an optional *postamble*. The preamble is prepended to each string within the braces, and the postamble is then appended to each resulting string:

```
$ echo a{d,c,b}e
ade ace abe
```

As this example demonstrates, the results of brace expansion are not sorted, as they are by filename expansion.

4.7. Process Substitution

On systems that can support it, Bash provides a facility known as *process substitution*. Process substitution is similar to command substitution in that its specification includes a command to execute, but the shell does not collect the command's output and insert it into the command line. Rather, Bash opens a pipe to the command, which is run in the background. The shell uses named pipes (FIFOs) or the */dev/fd* method of naming open files to expand the process substitution to a filename which connects to the pipe when opened. This filename becomes the result of the expansion. Process substitution can be used to compare the outputs of two different versions of an application as part of a regression test:

```
$ cmp <(old_prog) <(new_prog)
```

4.8. Prompt Customization

One of the more popular interactive features that Bash provides is the ability to customize the prompt. Both **$PS1** and **$PS2,** the primary and secondary prompts, are expanded before being displayed. Parameter and variable expansion is performed when the prompt string is expanded, so any shell variable can be put into the prompt (e.g., **$SHLVL**, which indicates how deeply the current shell is nested). Bash specially interprets characters in the prompt string preceded by a backslash. Some of these backslash escapes are replaced with the current time, the date, the current working directory, the username, and the command number or history number of the command being entered. There is even a backslash escape to cause the shell to change its prompt when running as root after an *su*. Before printing each primary prompt, Bash expands the variable **$PROMPT_COMMAND** and, if it has a value, executes the expanded value as a command, allowing additional prompt customization. For example, this assignment causes the current user, the current host, the time, the last component of the current working directory, the level of shell nesting, and the history number of the current command to be embedded into the primary prompt:

```
$ PS1='\u@\h [\t] \W($SHLVL:\!)\$ '
chet@odin [21:03:44] documentation(2:636)$ cd ..
chet@odin [21:03:54] src(2:637)$
```

The string being assigned is surrounded by single quotes so that if it is exported, the value of **$SHLVL** will be updated by a child shell:

```
chet@odin [21:17:35] src(2:638)$ export PS1
chet@odin [21:17:40] src(2:639)$ bash
chet@odin [21:17:46] src(3:696)$
```

The \$ escape is displayed as “**$**” when running as a normal user, but as “**#**” when running as root.

4.9. File System Views

Since Berkeley introduced symbolic links in 4.2 BSD, one of their most annoying properties has been the “warping” to a completely different area of the file system when using **cd**, and the resultant non-intuitive behavior of “**cd ..**”. The UNIX kernel treats symbolic links *physically*. When the kernel is translating a pathname in which one component is a symbolic link, it replaces all or part of the pathname while processing the link. If the contents of the symbolic link begin with a slash, the kernel replaces the pathname entirely; if not, the link contents replace the current component. In either case, the symbolic link is visible. If the link value is an absolute pathname, the user finds himself in a completely different part of the file system.

Bash provides a *logical* view of the file system. In this default mode, command and filename completion and builtin commands such as **cd** and **pushd** which change the current working directory transparently follow symbolic links as if they were directories. The **$PWD** variable, which holds the shell’s idea of the current working directory, depends on the path used to reach the directory rather than its physical location in the local file system hierarchy. For example:

```
$ cd /usr/local/bin
$ echo $PWD
/usr/local/bin
$ pwd
/usr/local/bin
$ /bin/pwd
/net/share/sun4/local/bin
$ cd ..
$ pwd
/usr/local
$ /bin/pwd
/net/share/sun4/local
$ cd ..
```

```
$ pwd
/usr
$ /bin/pwd
/usr
```

One problem with this, of course, arises when programs that do not understand the shell's logical notion of the file system interpret ".." differently. This generally happens when Bash completes filenames containing ".." according to a logical hierarchy which does not correspond to their physical location. For users who find this troublesome, a corresponding *physical* view of the file system is available:

```
$ cd /usr/local/bin
$ pwd
/usr/local/bin
$ set -o physical
$ pwd
/net/share/sun4/local/bin
```

4.10. Internationalization

One of the most significant improvements in version 1.13 of Bash was the change to "eight-bit cleanliness". Previous versions used the eighth bit of characters to mark whether or not they were quoted when performing word expansions. While this did not affect the majority of users, most of whom used only seven-bit ASCII characters, some found it confining. Beginning with version 1.13, Bash implemented a different quoting mechanism that did not alter the eighth bit of characters. This allowed Bash to manipulate files with "odd" characters in their names, but did nothing to help users enter those names, so version 1.13 introduced changes to readline that made it eight-bit clean as well. Options exist that force readline to attach no special significance to characters with the eighth bit set (the default behavior is to convert these characters to meta-prefixed key sequences) and to output these characters without conversion to meta-prefixed sequences. These changes, along with the expansion of keymaps to a full eight bits, enable readline to work with most of the ISO-8859 family of character sets, used by many European countries.

4.11. POSIX Mode

Although Bash is intended to be POSIX.2 conformant, there are areas in which the default behavior is not compatible with the standard. For users who wish to operate in a strict POSIX.2 environment, Bash implements a *POSIX mode*. When this mode is active, Bash modifies its default operation where it differs from POSIX.2 to match the standard. POSIX mode is entered when Bash is started with the **-posix** option. This feature is also available as an option to the **set** builtin, **set -o posix**. For compatibility with other GNU software that attempts to be POSIX.2 compliant, Bash also enters POSIX mode if the variable **$POSIXLY_CORRECT** is set when Bash is started or assigned a value during execution. **$POSIX_PEDANTIC** is accepted as well, to be compatible with some older GNU utilities. When Bash is started in POSIX mode, for example, it sources the file named by the value of **$ENV** rather than the "normal" startup files, and does not allow reserved words to be aliased.

5. New Features and Future Plans

There are several features introduced in the current version of Bash, version 1.14, and a number under consideration for future releases. This section will briefly detail the new features in version 1.14 and describe several features that may appear in later versions.

5.1. New Features in Bash-1.14

The new features available in Bash-1.14 answer several of the most common requests for enhancements. Most notably, there is a mechanism for including non-visible character sequences in prompts, such as those which cause a terminal to print characters in different colors or in standout mode. There was nothing preventing the use of these sequences in earlier versions, but the readline redisplay algorithm assumed each character occupied physical screen space and would wrap lines prematurely.

Readline has a few new variables, several new bindable commands, and some additional emacs mode default key bindings. A new history search mode has been implemented: in this mode, readline searches the history for lines beginning with the characters between the beginning of the current line and the cursor. The existing readline incremental search commands no longer match identical lines more than once. Filename completion now expands variables in directory names. The history expansion facilities are now nearly completely csh-compatible: missing modifiers have been added and history substitution has been extended.

Several of the features described earlier, such as **set -o posix** and **$POSIX_PEDANTIC**, are new in version 1.14. There is a new shell variable, **OSTYPE**, to which Bash assigns a value that identifies the version of UNIX it's running on (great for putting architecture-specific binary directories into the **$PATH**). Two variables have been renamed: **$HISTCONTROL** replaces **$history_control**, and **$HOSTFILE** replaces **$hostname_completion_file**. In both cases, the old names are accepted for backwards compatibility. The ksh *select* construct, which allows the generation of simple menus, has been implemented. New capabilities have been added to existing variables: **$auto_resume** can now take values of *exact* or *substring*, and **$HISTCONTROL** understands the value *ignoreboth*, which combines the two previously acceptable values. The **dirs** builtin has acquired options to print out specific members of the directory stack. The **$nolinks** variable, which forces a physical view of the file system, has been superseded by the **−P** option to the **set** builtin (equivalent to **set -o physical**); the variable is retained for backwards compatibility. The version string contained in **$BASH_VERSION** now includes an indication of the patch level as well as the "build version". Some little-used features have been removed: the **bye** synonym for **exit** and the **$NO_PROMPT_VARS** variable are gone. There is now an organized test suite that can be run as a regression test when building a new version of Bash.

The documentation has been thoroughly overhauled: there is a new manual page on the readline library and the *info* file has been updated to reflect the current version. As always, as many bugs as possible have been fixed, although some surely remain.

5.2. Other Features

There are a few features that I hope to include in later Bash releases. Some are based on work already done in other shells.

In addition to simple variables, a future release of Bash will include one-dimensional arrays, using the ksh implementation of arrays as a model. Additions to the ksh syntax, such as *varname*=(...) to assign a list of words directly to an array and a mechanism to allow the **read** builtin to read a list of values directly into an array, would be desirable. Given those extensions, the ksh **set −A** syntax may not be worth supporting (the **−A** option assigns a list of values to an array, but is a rather peculiar special case).

Some shells include a means of *programmable* word completion, where the user specifies on a per-command basis how the arguments of the command are to be treated when completion is attempted: as filenames, hostnames, executable files, and so on. The other aspects of the current Bash implementation could remain as-is; the existing heuristics would still be valid. Only when completing the arguments to a simple command would the programmable completion be in effect.

It would also be nice to give the user finer-grained control over which commands are saved onto the history list. One proposal is for a variable, tentatively named **HISTIGNORE**, which would contain a colon-separated list of commands. Lines beginning with these commands, after the restrictions of **$HISTCONTROL** have been applied, would not be placed onto the history list. The shell pattern-matching capabilities could also be available when specifying the contents of **$HISTIGNORE**.

One thing that newer shells such as **wksh** (also known as **dtksh**) provide is a command to dynamically load code implementing additional builtin commands into a running shell. This new builtin would take an object file or shared library implementing the "body" of the builtin (*xxx_builtin()* for those familiar with Bash internals) and a structure containing the name of the new command, the function to call when the new builtin is invoked (presumably defined in the shared object specified as an argument), and the documentation to be printed by the **help** command (possibly present in the shared object as well). It would manage the details of extending the internal table of builtins.

A few other builtins would also be desirable: two are the POSIX.2 **getconf** command, which prints the values of system configuration variables defined by POSIX.2, and a **disown** builtin, which causes a shell running with job control active to "forget about" one or more background jobs in its internal jobs table. Using **getconf**, for example, a user could retrieve a value for **$PATH** guaranteed to find all of the POSIX standard utilities, or find out how long filenames may be in the file system containing a specified directory.

There are no implementation timetables for any of these features, nor are there concrete plans to include them. If anyone has comments on these proposals, feel free to send me electronic mail.

6. Reflections and Lessons Learned

The lesson that has been repeated most often during Bash development is that there are dark corners in the Bourne shell, and people use all of them. In the original description of the Bourne shell, quoting and the shell grammar are both poorly specified and incomplete; subsequent descriptions have not helped much. The grammar presented in Bourne's paper describing the shell distributed with the Seventh Edition of UNIX† is so far off that it does not allow the command `who|wc`. In fact, as Tom Duff states:

> Nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help.‡

The POSIX.2 standard includes a *yacc* grammar that comes close to capturing the Bourne shell's behavior, but it disallows some constructs which sh accepts without complaint − and there are scripts out there that use them. It took a few versions and several bug reports before Bash implemented sh-compatible quoting, and there are still some "legal" sh constructs which Bash flags as syntax errors. Complete sh compatibility is a tough nut.

The shell is bigger and slower than I would like, though the current version is substantially faster than previously. The readline library could stand a substantial rewrite. A hand-written parser to replace the current *yacc*-generated one would probably result in a speedup, and would solve one glaring problem: the shell could parse commands in "$(...)" constructs as they are entered, rather than reporting errors when the construct is expanded.

As always, there is some chaff to go with the wheat. Areas of duplicated functionality need to be cleaned up. There are several cases where Bash treats a variable specially to enable functionality available another way (**$notify** vs. **set -o notify** and **$nolinks** vs. **set -o physical**, for instance); the special treatment of the variable name should probably be removed. A few more things could stand removal; the **$allow_null_glob_expansion** and **$glob_dot_filenames** variables are of particularly questionable value. The **$[...]** arithmetic evaluation syntax is redundant now that the POSIX-mandated **$((...))** construct has been implemented, and could be deleted. It would be nice if the text output by the **help** builtin were external to the shell rather than compiled into it. The behavior enabled by **$command_oriented_history**, which causes the shell to attempt to save all lines of a multi-line command in a single history entry, should be made the default and the variable removed.

7. Availability

As with all other GNU software, Bash is available for anonymous FTP from *prep.ai.mit.edu:/pub/gnu* and from other GNU software mirror sites. The current version is in *bash-1.14.1.tar.gz* in that directory. Use *archie* to find the nearest archive site. The latest version is always available for FTP from *bash.CWRU.Edu:/pub/dist.* Bash documentation is available for FTP from *bash.CWRU.Edu:/pub/bash.*

The Free Software Foundation sells tapes and CD-ROMs containing Bash; send electronic mail to `gnu@prep.ai.mit.edu` or call `+1-617-876-3296` for more information.

Bash is also distributed with several versions of UNIX-compatible systems. It is included as /bin/sh and /bin/bash on several Linux distributions (more about the difference in a moment), and as contributed

---

†S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell", *Bell System Technical Journal*, 57(6), July-August, 1978, pp. 1971-1990.

‡Tom Duff, "Rc − A Shell for Plan 9 and UNIX systems", *Proc. of the Summer 1990 EUUG Conference*, London, July, 1990, pp. 21-33.

software in BSDI's BSD/386* and FreeBSD.

The Linux distribution deserves special mention. There are two configurations included in the standard Bash distribution: a "normal" configuration, in which all of the standard features are included, and a "minimal" configuration, which omits job control, aliases, history and command line editing, the directory stack and **pushd/popd/dirs,** process substitution, prompt string special character decoding, and the *select* construct. This minimal version is designed to be a drop-in replacement for the traditional UNIX /bin/sh, and is included as the Linux /bin/sh in several packagings.

8. Conclusion

Bash is a worthy successor to sh. It is sufficiently portable to run on nearly every version of UNIX from 4.3 BSD to SVR4.2, and several UNIX workalikes. It is robust enough to replace sh on most of those systems, and provides more functionality. It has several thousand regular users, and their feedback has helped to make it as good as it is today – a testament to the benefits of free software.

*BSD/386 is a trademark of Berkeley Software Design, Inc.