Bash, the Bourne−Again Shell

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

ABSTRACT

An overview of /bin/sh−compatible shells is presented, as well as an introduction to the POSIX.2 shell and tools standard. These serve as an introduction to bash. A comparison of bash to sh and ksh is presented, as well as a discussion of features unique to bash. Finally, some of the changes and new features to appear in the next bash release will be discussed.

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 [1].

Bash is an **sh**−compatible shell that incorporates useful features from the Korn shell (**ksh**) [2] and the C shell (**csh**) [3], described later in this article. It is ultimately intended to be a conformant implementation of the IEEE POSIX Shell and Tools 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 is a shell?

At its base, a shell is simply a macro processor that executes commands. A UNIX shell is both a command interpreter, which provides the user interface to the rich set of UNIX utilities, and a programming language, allowing these utilitites to be combined. The shell reads commands either from a terminal or a file. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands in directories like **/bin**, allowing users or groups to establish custom environments.

2.1. Command Interpreter

A shell allows execution of UNIX commands, both synchronously and asynchronously. The *redirection* constructs permit fine-grained control of the input and output of those commands, and the shell allows control over the contents of their environment. UNIX shells also provide a small set of built-in commands (*builtins*) implementing functionality impossible (e.g., **cd**, **break**, **continue**, and **exec**) or inconvenient

† UNIX is a trademark of X/OPEN

(**history**, **getopts**, **kill**, or **pwd**, for example) to obtain via separate utilities. Shells may be used interactively or non-interactively: they accept input typed from the keyboard or from a file.

2.2. Programming Language

While executing commands is essential, most of the power (and complexity) of shells is due to their embedded programming languages. Like any high-level language, the shell provides variables, flow control constructs, quoting, and functions.

The basic syntactic element is a *simple command*. A simple command consists of an optional set of variable assignments, a command word, and an optional list of arguments. Operators to redirect input and output may appear anywhere in a simple command. Some examples are:

```
who
trn -e -S1 -N
ls -l /bin > binfiles
make > make.out 2>make.errs
```

A pipeline is a sequence of two or more commands separated by the character |. The standard output of the first command is connected to the standard input of the second. Examples of pipelines include:

```
who | wc -l
ls -l | sort +3nr
```

Simple commands and pipelines may be combined into *lists*. A list is a sequence of pipelines separated by one of **;**, **&**, **&&**, or ||, and optionally terminated by **;**, **&**, or a newline. Commands separated by **;** are executed sequentially; the shell waits for each to complete in turn. If a command is terminated by **&**, the shell executes it in the *background*, and does not wait for it to finish. If two commands are separated by **&&**, the second command executes only if the first command succeeds. A separator of || causes the second command to execute only if the first fails. Some examples are:

```
who ; date
cd /usr/src || exit 1
cd "$@" && xtitle $HOST: $PWD
```

The shell programming language provides a variety of flow control structures. The **for** command allows a list of commands to be executed once for each word in a word list. The **case** command allows a list to be executed if a word matches a specified pattern. The **while** and **until** commands execute a list of commands as long as a guard command completes successfully or fails, respectively. The **if** command allows execution of different command lists depending on the exit status of a guard command.

A *shell function* associates a list of commands with a name. Each time the name is used as a simple command, the list is executed. This execution takes place in the current shell context; no new process is created. Functions may have their own argument lists and local variables, and may be recursive.

The shell language provides variables, which may be both set and referenced. A number of special parameters are present, such as **$@**, which returns the shell's positional parameters (command-line arguments), **$?**, the exit status of the previous command, and **$$**, the shell's process I.D. In addition to providing special parameters and user-defined variables, the shell permits the values of certain variables to control its behavior. Some of these variables include **IFS**, which controls how the shell splits words, **PATH**, which tells the shell where to look for commands, and **PS1**, whose value is the string the shell uses to prompt for commands. There are a few variables whose values are set by the shell and normally only referenced by users; **PWD**, whose value is the pathname of the shell's current working directory, is one such variable. Variables can be used in nearly any shell context and are particularly valuable when used with control structures.

There are several shell *expansions*. A variable *name* is expanded to its value using ${*name*}, where the braces are optional. There are a number of parameter expansions available. For example, there are ${*name*:−*word*}, which expands to *word* if *name* is unset or null, and the inverse ${*name*:+*word*}, which expands to *word* if *name* is set and not null. *Command substitution* allows the output of a command to replace the command name. The syntax is ‘*command*‘. *Command* is executed and it and the backquotes are replaced by its output, with trailing newlines removed. *Pathname expansion* is a way to expand a word to a set of filenames. Words are regarded as patterns, in which the characters *****, **?**, and **[** have special meaning. Words containing these special characters are replaced with a sorted list of matching pathnames. If a word generates no matches, it is left unchanged.

Quoting is used to remove the special meaning of characters or words. It can disable special treatment for shell operators or other special characters, prevent reserved words from being recognized as such, and inhibit variable expansion. The shell has three quoting mechanisms: a backslash preserves the literal value of the next character, a pair of single quotes preserves the literal value of each character between the quotes, and a pair of double quotes preserves the literal meaning of enclosed characters while allowing some expansions.

Some of the commands built into the shell are part of the programming language. The **break** and **continue** commands control loop execution as in the C language. The **eval** builtin allows a string to be parsed and executed as a command. **Wait** tells the shell to pause until the processes specified as arguments have exited.

2.3. Interactive Features

Shells have begun offering features geared specifically for interactive use rather than to augment the programming language. These interactive features include job control, command line editing, history and aliases.

Job control is a facility provided jointly by the shell and the UNIX kernel that allows users to selectively stop (suspend) and restart (resume) processes. Each pipeline executed by the shell is referred to as a *job*. Jobs may be suspended and restarted in either the foreground, where they have access to the terminal, or background, where they are isolated and cannot read from the terminal. Typing the *suspend* character while a process is running stops that process and returns control to the shell. Once a job is suspended, the user manipulates the job's state, using **bg** to continue it in the background, **fg** to return it to the foreground and await its completion, or **kill** to send it a signal. The **jobs** command lists the status of jobs, and **wait** will pause the shell until a specified job terminates. The shell provides a number of ways to refer to a job, and will notify the user whenever a background job terminates.

With the advent of more powerful terminals and terminal emulators, more sophisticated interaction than that provided by the UNIX kernel terminal driver is possible. Some shells offer command line editing, which permits a user to edit lines of input using familiar *emacs* or *vi*-style commands before submitting them to the shell. Editors allow corrections to be made without having to erase back to the point of error, or start the line anew. Command line editors run the gamut from a small fixed set of commands and key bindings to input facilities which allow arbitrary actions to be bound to a key or key sequence.

Modern shells also keep a history, which is the list of commands a user has typed. Shell facilities are available to recall previous commands and use portions of old commands when composing new ones. The command history can be saved to a file and read back in at shell startup, so it persists across sessions. Shells which provide both command editing and history generally have editing commands to interactively step forward and backward through the history list.

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"`).

3. The POSIX Shell Standard

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 Tools standard has been developed by IEEE Working Group 1003.2 (POSIX.2)[4]. 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;
- 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, command line editing, and history. Interestingly enough, only *vi*-style line editing commands have been standardized; *emacs* editing commands were left out due to objections.

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. 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.

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* .

4. Shell Comparison

This section compares features of bash, sh, and ksh (the three shells closest to POSIX compliance). Since ksh and bash are supersets of sh, the features common to all three are covered first. Some of the features bash and ksh contain which are not in sh will be discussed. Next, features unique to bash will be listed. The first three sections provide a progressively more detailed overview of bash. Finally, features of ksh-88 (the currently-available version) not in sh or bash will be presented.

4.1. Common Features

All three shells have the same basic feature set, which is essentially that provided by sh and described in any sh manual page. Bash and ksh are both sh supersets, and so all three provide the command interpreter and programming language described earlier. The shell grammar, syntax, flow control, redirections, and builtins implemented by the Bourne shell are the baseline for subsequent discussion.

4.2. Features in bash and ksh

Ksh and bash have several features in common beyond this base level of functionality. Some of this is due to the POSIX.2 standard. Other functions have been implemented in bash using ksh as a guide.

4.2.1. Variables and Variable Expansion

Bash and ksh have augmented variable expansion. Arithmetic substitution allows an expression to be evaluated and the result substituted. 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
```

Variables may be declared as *integer*, which causes arithmetic evaluation to be performed on the value whenever they are assigned to.

There are new expansions to obtain the length of a variable's value and to remove substrings matching specified patterns from the beginning and end of variable values. A new form of command substitution, **$(***list***)**, is much easier to nest than '*list*' and has simplified quoting rules.

There are new variables to control the shell's behavior, and additional variables set or interpreted specially by the shell. **RANDOM** and **SECONDS** are *dynamic* variables: their values are generated afresh each time they are referenced. **RANDOM** returns a different random number each time it is referenced, and **SECONDS** returns the number of seconds since the shell was started or the variable was assigned to, plus any value assigned. **PWD** and **OLDPWD** are set to the current and previous working directories, respectively. **TMOUT** controls how long the shell will wait at a prompt for input. If **TMOUT** is set to a value greater than zero, the shell exits after waiting that many seconds for input. **REPLY** is the default variable for the **read** builtin; if no variable names are supplied as arguments, the line read is assigned to **REPLY**.

4.2.2. New and Modified Builtins

Both shells expand the basic sh set of builtin commands. **Let** provides a way to perform arithmetic on shell variables. Shell programmers use **typeset** (bash includes **declare** as a synonym) to assign attributes such as *export* and *readonly* to variables. **Getopts** is used by shell scripts to parse script options and arguments. The **set** command has a new option **–o** which takes option names as arguments. Option names are synonyms for the other set options (e.g., **–f** and **–o noglob**) or provide new functionality (**-o notify**, for example). The **read** builtin takes a new **–r** option to specify that a line ending in a backslash should not be continued.

4.2.3. Tilde Expansion

Tilde expansion is a feature adopted from the C shell. A tilde character at the beginning of a word is expanded to either **$HOME** or the home directory of another user, depending on what follows the tilde.

4.2.4. Interactive Improvements

The most noticable improvements over sh are geared for interactive use. Ksh and bash provide job control in a very similar fashion, with the same options to enable and disable it (**set -o monitor**) and the same builtin commands to manipulate jobs (**jobs/fg/bg/kill/wait**).

Command line editing, with emacs and vi-style key bindings, is available in both shells. The Bash *readline* library is considerably more sophisticated than the ksh editing library: it allows arbitrary key bindings, macros, a per-user customization file (*~/.inputrc*), a number of variables to further customize behavior, and a much larger set of bindable editing commands. The ksh editing library provides a small fixed command set and only clumsy macros.

Both shells offer access to the command history. The in-line editing options have default key bindings to access the history list. The **fc** command is provided to re-execute previous commands and display the contents of the history list.

4.2.5. Miscellaneous Changes and Improvments

Other improvements include aliases, the **select** shell language construct, which supports the generation and presentation of simple menus, and extensions to the **export** and **readonly** builtins which allow variables to be assigned values at the same time the attributes are set. Word splitting has changed: if two or more adjacent word splitting characters occur, bash and ksh will generate null fields; sh makes runs of multiple field separator charactors the same as a single separator. Bash and ksh split only the results of expansion, rather than every word as sh does, closing a long-standing shell security hole.

Shell functions in bash and ksh may have local variables. Variables declared with **typeset** (or the bash synonym, **local**), have a scope restricted to the function and its descendents, and may shadow variables defined by the invoking shell. Local variables are removed when a function completes.

4.3. Features Unique to bash

Naturally, bash includes features not in sh or ksh. This section discusses some 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.

4.3.1. 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.

4.3.2. 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.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 "$@" && 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.

4.3.3. 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 his 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: here are bindings to 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 can 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.

4.3.4. 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.

4.3.5. 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; **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**); 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 a suspended job with that name before starting a new process.

4.3.6. Brace Expansion

Since sh offers no convenient way to generate arbitrary strings that share a common prefix or suffix (pathname expansion requires that the filenames exist), bash implements *brace expansion*, a capability picked up from csh. Brace expansion is similar to pathname 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
```

4.3.7. 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 the value of 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, **SHLVL** will be updated by a child shell:

```
chet@odin [21:13:35] 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.

4.3.8. POSIX Mode

Although bash is intended to be POSIX.2 compliant, 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 **-o posix** option or when **set -o posix** is executed. For compatibility with other GNU software that attempts to be POSIX.2 compliant, bash also enters POSIX mode if either of the variables **POSIX_PEDANTIC** or **POSIXLY_CORRECT** is set when bash is started or assigned a value during execution. When bash is started in POSIX mode, for example, the **kill** builtin’s **–l** option behaves differently: it lists the names of all signals on a single line separated by spaces, rather than listing the signal names and their corresponding numbers.

Some of the default bash behavior differs from other shells as a result of the POSIX standard. For instance, bash includes the **!** reserved word to negate the return status of a pipeline because it has been defined by POSIX.2. Neither sh nor ksh has implemented that feature.

4.4. Features Unique to ksh

Ksh includes a number of features not in the currently-released version of bash, version 1.14. Unless noted, none of these features is in the POSIX.2 standard. Where appropriate the equivalent bash features are noted.

4.4.1. The ksh Language

A new compound command folds **test** into the ksh language, delimited by the reserved words **[[** and **]]**. The syntax is identical to **test** with a few changes: for instance, instead of **–a** and **–o**, **&&** and || are used. The words between **[[** and **]]** are not processed for word splitting or filename generation. The new command does pattern matching as well as string comparison, a la the **case** command. This new control structure does have the advantage of reducing common argument problems encountered using test (e.g. **test "$string"**, where **$string** expands to **–f**), but at the cost of bloating the language. The POSIX.2 test algorithm that bash uses, along with some programmer care, alleviates those problems in a backwards-compatible way with no additions to the language. The one capability of **[[ ]]** not available in bash is its ability to test whether an individual **set –o** option is turned on or off.

Other parts of the ksh language are not common to bash. The **((...))** operator, equivalent to **let "..."**, is unique to ksh, as are the concept of co-processes and the **time** keyword to time commands and pipelines.

4.4.2. Functions and Aliases

The Korn shell has *autoloaded* functions. A function marked as *autoload* is not defined until it is first executed. When such a function is executed, a search is made through the directories in **FPATH** (a colon-separated list of directories similar to **PATH**) for a file with the same name as the function. That file is then read in as with the **.** command; presumably the function is defined therein. There is a pair of shell functions included in the bash distribution (*examples/functions/autoload*) that provide much of this functionality without changing the shell itself.

Ksh functions are scoped in such a way that the environment in which they are executed is closer to a shell script environment. Bash uses the POSIX.2 scoping rules, which make the function execution environment an exact copy of the shell environment with the replacement of the shell's positional paramters with the function arguments. Korn shell functions do not share options or traps with the invoking shell.

Ksh has *tracked* aliases, which alias a command name to its full pathname. Bash has true command hashing.

4.4.3. Arrays

Arrays are an aspect of ksh that has no real bash equivalent. They are easy to create and manipulate: an array is created automatically by using subscript assignment (**name**[*index*]=**value**), and any variable may be referred to as an array. Ksh arrays, however, have several annoying limitations: they may be indexed only up to 512 or 1024 elements, depending on how the shell is compiled, and there is only the clumsy **set -A** to assign a list of values sequentially. Despite these limits, arrays are useful, if underutilized by shell programmers.

4.4.4. Builtin Commands

Some of the builtin commands have been extended or are new in ksh. The **print** builtin was included to work around the incompatibilities and limitations of **echo**. The **whence** command tells what would happen if each argument were typed as a command name. The **cd** builtin has been extended to take up to two arguments: if two arguments are supplied, the second is substituted for the first in the current directory name and the shell changes to the resultant directory name. The ksh **trap** builtin accepts **ERR** and **DEBUG** as trap names. The **ERR** trap is executed when a command fails; **DEBUG** is executed after every simple command.

The bash distribution includes shell functions that implement **print** and **whence** and the extensions to **cd**.

4.4.5. Expansion

The ksh filename generation (*globbing*) facilities have been extended beyond their bash and sh counterparts. In this area, ksh can be thought of as *egrep* to the bash *grep*. Ksh globbing offers things like alternation, the ability to match zero or more instances of a pattern, and the ability to match exactly one occurrence of any of a list of patterns.

4.4.6. Startup Files

Ksh and bash execute startup files differently. Ksh expands **ENV** and sources the file it names for every shell. Bash sources **$ENV** only in non-interactive shells; interactive shells source fixed files, as explained in the previous section. The POSIX standard has specified the ksh behavior, so bash acts the same as ksh if started with the **−posix** or **−o posix** options.

4.4.7. History

Finally, the ksh history implementation differs slightly from bash. Each instance of bash keeps the history list in memory and offers options to the **history** builtin to write the list to or read it from a named file. Ksh keeps the history in a file, which it accesses each time a command is saved to or retrieved from the history. Ksh history files may be shared among different concurrent instances of ksh, which could be a benefit to the user.

5. Features in Bash-2.0

The next release of bash, 2.0, will be a major overhaul. It will include many new features, for both programming and interactive use. Redundant existing functions will be removed. There are several cases where bash treats a variable specially to enable functionality available another way (**$nolinks** vs. **set -o physical**, for example); the special treatment of the variable name will be removed.

5.1. Arrays

Bash-2.0 will include arrays which are a superset of those in ksh, with the size limitations removed. The **declare**, **readonly**, and **export** builtins will accept options to specify arrays, and the **read** builtin will have an option to read a list of words and assign them directly to an array. There will also be a new array *compound assignment* syntax available for assignment statements and the **declare** builtin. This new syntax has the form *name*=(*value1* ... *valueN*), where each *value* has the form [*subscript*]=*string*. Only the *string* is required. If the optional brackets and *subscript* are included, that index is assigned to, otherwise the index of the element assigned is the last index assigned to by the statement plus one. Indexing starts at zero. The same syntax is accepted by **declare**. Individual array elements may be assigned to using the ksh *name*[*subscript*]=*value*.

5.2. Dynamic Loading

On systems that support the *dlopen*(3) library function, bash-2.0 will allow new builtins to be loaded into a running shell from a shared object file. The new builtins will have access to the rest of the shell facilities, but programmers will be subject to a few structural rules. This will be provided via a new option to **enable**.

5.3. Builtins

Some of the existing builtins will change in bash-2.0. As previously noted, **declare**, **export**, **readonly**, and **read** will accept new options to specify arrays. The **jobs** builtin will be able to list only stopped or running jobs. The **enable** command will take a new **−s** option to restrict its actions to the POSIX.2 *special* builtins. **Kill** will be able to list signal numbers corresponding to individual signal names. The readline library interface, **bind**, will have an option to remove the binding for any key sequence (which is not the same as binding it to self-insert).

There will be two new builtin commands in bash-2.0. The **disown** command will remove jobs from bash's internal jobs table when job control is active. A disowned job will not be listed by the jobs command, nor will its exit status be reported. Disowned jobs will not be sent a **SIGHUP** when an interactive shell exits. Most of the shell's optional or *toggled* functionality will be folded into the new **shopt** builtin. Many of the variables which alter the shell's behavior when set (regardless of their value) will be made options settable with **shopt**. Examples of such variables include **allow_null_glob_expansion**, **glob_dot_filenames**, and **MAIL_WARNING**.

5.4. Variables and Variable Expansion

Bash-2.0 will implement several new variable expansions. These will answer several of the most persistant requests for new features. It will be possible to "indirectly reference" a variable with an expansion, like using `eval \$${name}` to reference a variable named by `${name}`. Expansions will be available to retrieve substrings of variables in an *awk*-like manner: starting at a specific index, retrieving some number of characters or the rest of the string. It will be possible to retrieve sequences of array elements like this, too. It would be nice to have a way to replace portions of a variable matching a pattern the same way leading or trailing substrings are presently stripped; that capability may be available.

Another new expansion will provide a way to create strings containing arbitrary characters, which is inconvenient in the current version. Words of the form $'*string*' will expand to *string* with backslash-escaped characters in *string* replaced as specified by the ANSI C standard. As with other single-quoted shell strings, the only character that may not appear in *string* is a single quote.

The shell variables will change also. A new variable **HISTIGNORE** will supersede **HISTCONTROL**. **HISTIGNORE** is the history analogy of **FIGNORE**: a colon-separated list of patterns specifying commands to omit from the history list. The special pattern '&' will match the previous history line, to provide the **HISTCONTROL** *ignoredups* behavior. Many variables which modify the shell's behavior will lose their special meaning. Variables such as **notify** and **noclobber** which provide functionality available via other mechanisms will no longer be treated specially. Other variables will be folded into **shopt**. The **history_control** and **hostname_completion_file** variables, superseded by **HISTCONTROL** and **HOSTFILE** respectively, will be removed.

5.5. Readline

Naturally, there will be improvements to readline as well. All of the POSIX.2 *vi*-mode editing commands will be implemented; missing commands like 'm' to save the current cursor position (*mark*) and the '@' command for macro expansion will be available. The ability to set the mark and exchange the current cursor position (*point*) and mark will be added to the readline emacs mode as well. Since there are commands to set the mark, commands to manipulate the region (the characters between the point and the mark) will be available. Commands have been added to the readline emacs mode for more complete ksh compatibility, such as the C-]*c* character search command.

5.6. Configuration

Bash was the first GNU program to completely autoconfigure. Its autoconfiguration mechanism predates *autoconf*, the current GNU configuration program, and needs updating. Bash-2.0 may include an autoconf-based configuration script, if necessary new functionality can be added to autoconf, or its limitations bypassed.

5.7. Miscellaneous

The POSIX mode will be improved in bash-2.0; it will provide a more complete superset of the POSIX standard. For the first time, bash will recognize the existance of the POSIX.2 *special* builtins.

A new trap value, **DEBUG**, will be present, as in ksh. Commands specified with a **DEBUG** trap will be executed after every simple command. Since this makes shell script debuggers possible, I hope to include a bash debugger in the bash-2.0 release.

6. Availability

The current version of bash is available for anonymous FTP from prep.ai.mit.edu as */pub/gnu/bash-1.14.2.tar.gz*.

7. Conclusion

This paper has presented an overview of bash, compared its features with those of other shells, and hinted at features in the next release, bash-2.0.

Bash is a solid replacement for sh. It is sufficiently portable to run on nearly every version of UNIX from 4.3 BSD to SVR4.2, and several UNIX workalikes, and robust enough to replace sh on most of those systems, It is very close to POSIX.2-conformant in POSIX mode, and is getting faster. It is not, unfortunately, getting smaller, but there are many optional features. It is very easy to build a small subset to use as a direct replacement for /bin/sh.

Bash has thousands of users worldwide, all of whom have helped to make it better. Another testament to the benefits of free software.

8. References

[1] S. R. Bourne, ‘‘UNIX Time-Sharing System: The UNIX Shell’’, *Bell System Technical Journal*, 57(6), July-August, 1978, pp. 1971-1990.

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

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

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

9. Author Information

Chet Ramey is a software engineer working at Case Western Reserve University. He has a B.S. in Computer Engineering and an M.S. in Computer Science, both from CWRU. He has been working on bash for six years, and the primary maintainer for one.