1.2. Advantages of the Bourne Again SHell

1.2.1. Bash is the GNU shell

The GNU project (GNU's Not UNIX) provides tools for UNIX-like system administration which are free software and comply to UNIX standards.

Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use; these include command line editing, unlimited size command history, job control, shell functions and aliases, indexed arrays of unlimited size, and integer arithmetic in any base from two to sixty-four. Bash can run most sh scripts without modification.

Like the other GNU projects, the bash initiative was started to preserve, protect and promote the freedom to use, study, copy, modify and redistribute software. It is generally known that such conditions stimulate creativity. This was also the case with the bash program, which has a lot of extra features that other shells can't offer.

1.2.2. Features only found in bash

1.2.2.1. Invocation

In addition to the single-character shell command line options which can generally be configured using the **set** shell built-in command, there are several multi-character options that you can use. We will come across a couple of the more popular options in this and the following chapters; the complete list can be found in the Bash info pages, Bash features->Invoking Bash.

1.2.2.2. Bash startup files

Startup files are scripts that are read and executed by Bash when it starts. The following subsections describe different ways to start the shell, and the startup files that are read consequently.

1.2.2.2.1. Invoked as an interactive login shell, or with `--login'

Interactive means you can enter commands. The shell is not running because a script has been activated. A login shell means that you got the shell after authenticating to the system, usually by giving your user name and password.

Files read:

- `/etc/profile`
- `~/.bash_profile`, `~/.bash_login` or `~/.profile`: first existing readable file is read
- `~/.bash_logout` upon logout.

Error messages are printed if configuration files exist but are not readable. If a file does not exist, bash searches for the next.

1.2.2.2.2. Invoked as an interactive non-login shell

A non-login shell means that you did not have to authenticate to the system. For instance, when you open a terminal using an icon, or a menu item, that is a non-login shell.

Files read:

- `~/.bashrc`

This file is usually referred to in `~/.bash_profile`:

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

See Chapter 7 for more information on the **if** construct.

1.2.2.2.3. Invoked non-interactively

All scripts use non-interactive shells. They are programmed to do certain tasks and cannot be instructed to do other jobs than those for which they are programmed.

Files read:

- defined by `BASH_ENV`

`PATH` is not used to search for this file, so if you want to use it, best refer to it by giving the full path and file name.

1.2.2.2.4. Invoked with the sh command

Bash tries to behave as the historical Bourne **sh** program while conforming to the POSIX standard as well.

Files read:

- `/etc/profile`
- `~/.profile`

When invoked interactively, the `ENV` variable can point to extra startup information.

1.2.2.2.5. POSIX mode

This option is enabled either using the **set** built-in:

set `-o posix`

or by calling the **bash** program with the `--posix` option. Bash will then try to behave as compliant as possible to the POSIX standard for shells. Setting the `POSIXLY_CORRECT` variable does the same.

Files read:

- defined by `ENV` variable.

1.2.2.2.6. Invoked remotely

Files read when invoked by **rshd**:

- `~/.bashrc`

Avoid use of r-tools

Be aware of the dangers when using tools such as **rlogin**, **telnet**, **rsh** and **rcp**. They are intrinsically insecure because confidential data is sent over the network unencrypted. If you need tools for remote execution, file transfer and so on, use an implementation of Secure SHell, generally known as SSH, freely available from http://www.openssh.org. Different client programs are available for non-UNIX systems as well, see your local software mirror.

1.2.2.2.7. Invoked when UID is not equal to EUID

No startup files are read in this case.

1.2.2.3. Interactive shells

1.2.2.3.1. What is an interactive shell?

An interactive shell generally reads from, and writes to, a user's terminal: input and output are connected to a terminal. Bash interactive behavior is started when the **bash** command is called upon without non-option arguments, except when the option is a string to read from or when the shell is invoked to read from standard input, which allows for positional parameters to be set (see Chapter 3).

1.2.2.3.2. Is this shell interactive?

Test by looking at the content of the special parameter `-`, it contains an 'i' when the shell is interactive:

```
eddy:~> echo $-
himBH
```

In non-interactive shells, the prompt, `PS1`, is unset.

1.2.2.3.3. Interactive shell behavior

Differences in interactive mode:

- Bash reads startup files.
- Job control enabled by default.
- Prompts are set, `PS2` is enabled for multi-line commands, it is usually set to ">". This is also the prompt you get when the shell thinks you entered an unfinished command, for instance when you forget quotes, command structures that cannot be left out, etc.
- Commands are by default read from the command line using **readline**.

- Bash interprets the shell option `ignoreeof` instead of exiting immediately upon receiving EOF (End Of File).
- Command history and history expansion are enabled by default. History is saved in the file pointed to by `HISTFILE` when the shell exits. By default, `HISTFILE` points to `~/.bash_history`.
- Alias expansion is enabled.
- In the absence of traps, the `SIGTERM` signal is ignored.
- In the absence of traps, `SIGINT` is caught and handled. Thus, typing **Ctrl**+**C**, for example, will not quit your interactive shell.
- Sending `SIGHUP` signals to all jobs on exit is configured with the `huponexit` option.
- Commands are executed upon read.
- Bash checks for mail periodically.
- Bash can be configured to exit when it encounters unreferenced variables. In interactive mode this behavior is disabled.
- When shell built-in commands encounter redirection errors, this will not cause the shell to exit.
- Special built-ins returning errors when used in POSIX mode don't cause the shell to exit. The built-in commands are listed in Section 1.3.2.
- Failure of **exec** will not exit the shell.
- Parser syntax errors don't cause the shell to exit.
- Simple spell check for the arguments to the **cd** built-in is enabled by default.
- Automatic exit after the length of time specified in the `TMOUT` variable has passed, is enabled.

More information:

- Section 3.2
- Section 3.6
- See Chapter 12 for more about signals.
- Section 3.4 discusses the various expansions performed upon entering a command.

1.2.2.4. Conditionals

Conditional expressions are used by the **[[** compound command and by the **test** and **[** built-in commands.

Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. You only need one object, for instance a file, to do the operation on.

There are string operators and numeric comparison operators as well; these are binary operators, requiring two objects to do the operation on. If the `FILE` argument to one of the primaries is in the form `/dev/fd/N`, then file descriptor N is checked. If the `FILE` argument to one of the primaries is one of `/dev/stdin`,

`/dev/stdout` or `/dev/stderr`, then file descriptor 0, 1 or 2 respectively is checked.

Conditionals are discussed in detail in Chapter 7.

More information about the file descriptors in Section 8.2.3.

1.2.2.5. Shell arithmetic

The shell allows arithmetic expressions to be evaluated, as one of the shell expansions or by the **let** built-in.

Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence and associativity are the same as in the C language, see Chapter 3.

1.2.2.6. Aliases

Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the **alias** and **unalias** commands.

Bash 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. The commands following the alias definition on that line are not affected by the new alias.

Aliases are expanded when a 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.

We will discuss aliases in detail in Section 3.5.

1.2.2.7. Arrays

Bash provides one-dimensional array variables. Any variable may be used as an array; the **declare** built-in will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Arrays are zero-based. See Chapter 10.

1.2.2.8. Directory stack

The directory stack is a list of recently-visited directories. The **pushd** built-in adds directories to the stack as it changes the current directory, and the **popd** built-in removes specified directories from the stack and changes the current directory to the directory removed.

Content can be displayed issuing the **dirs** command or by checking the content of the `DIRSTACK` variable.

More information about the workings of this mechanism can be found in the Bash info pages.

1.2.2.9. The prompt

Bash makes playing with the prompt even more fun. See the section *Controlling the Prompt* in the Bash info pages.

1.2.2.10. The restricted shell

When invoked as **rbash** or with the `--restricted` or `-r` option, the following happens:

- The **cd** built-in is disabled.
- Setting or unsetting `SHELL`, `PATH`, `ENV` or `BASH_ENV` is not possible.
- Command names can no longer contain slashes.
- Filenames containing a slash are not allowed with the **.** (**source**) built-in command.
- The **hash** built-in does not accept slashes with the `-p` option.
- Import of functions at startup is disabled.
- `SHELLOPTS` is ignored at startup.
- Output redirection using >, >|, ><, **>&**, **&>** and >> is disabled.
- The **exec** built-in is disabled.
- The `-f` and `-d` options are disabled for the **enable** built-in.
- A default `PATH` cannot be specified with the **command** built-in.
- Turning off restricted mode is not possible.

When a command that is found to be a shell script is executed, **rbash** turns off any restrictions in the shell spawned to execute the script.

More information:

- Section 3.2
- Section 3.6
- Info Bash->Basic Shell Features->Redirections
- Section 8.2.3: advanced redirection
