Work in progress

Shells

A shell (also called *a command line interpreter*) is a program that provides a command line interface, a bridge between a user and the operating system. When a user logs in, the shell specified in the user's profile loads up and greets the user with a welcoming command prompt. The default shell in Slackware is Bash (Bourne-again shell). Other popular shells that Slackware ships with include:

- Korn Shell (ksh)
- C Shell (tcsh)
- Z Shell (zsh)

Although we are going to focus exclusively on `bash`, one can switch and try out other shells:

```
root@darkstar:~# chsh -s /bin/zsh fred
```

This will change the login shell to `zsh` for the user `fred`. If we do not want to modify a user's default shell, we can just start a given shell by typing:

```
user@darkstar:~$ zsh
```

To learn more about those shells please see this article.

Bash and its Startup Files

Being released in 1989, `Bash` predates Slackware by a few years and has been part of the distro since its early days. Slackware 1.01 featured Bash-1.12.3. Over the years Bash has developed a wide range of features and has become the default shell in many distros including Slackware.

Login & Interactive Shells

To understand how shell initialisation files are loaded we need to introduce the concept of *login* and *interactive* shells.

In a nutshell (no pun intended), a `login shell` is the one that starts when a user logs in providing their username and password (also remotely using ***ssh***). Technically speaking, a login shell is one *whose first character of argument zero is a - or, or one started with the `--login` option* (Source: Bash manual). An `interactive shell` is one which a user can interact with by typing commands on the command prompt. Please note that although some shell scripts allow for user interaction, the shell they start is non-interactive. The following are some common ways of shell invocation:

Shell Invocation	Login	Non-login	Interactive	Non-interactive
Login at runlevel 3 (Slackware default login)	+	-	+	-

Shell Invocation	Login	Non-login	Interactive	Non-interactive
Login remotely (ssh)	+	-	+	-
Opening a terminal window	-	+	+	-
Opening a terminal window (bash -l[1])	+	-	+	-
Shell script (#!/bin/bash)	-	+	-	+
Remote command (ssh host command)	-	+	-	+
Switch user (su)	-	+	+	-
Switch user (su -)	+	-	+	-

You can tell whether a shell is a login one by checking the value of the `$0` variable. If it starts with a dash (`-`), it is a login one:

```
user@darkstar:~$ echo $0
-bash
```

The distinction between `login` and `non-login` shells is crucial in order to understand which startup files are read when a shell starts, which, in turn, determines which environment variables are set. A good example illustrating this problem is the $PATH variable:

su (non-login shell)

```
root@darkstar:/home/user# echo $PATH
/usr/local/sbin:/usr/local/bin:/sbin:/usr/sbin:/bin:/usr/bin
```

su - or **su -l** (login shell)

```
root@darkstar:~# echo $PATH
/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr
/lib/java/bin:/usr/lib/kde4/libexec:/usr/lib/qt/bin:/usr/share/texmf/bin
```

For that reason, it is recommended to start a login shell with (**su -**) when building SlackBuild scripts.

Login Shell Initialisation

Examples: Slackware login (runlevel 3), an interactive shell started with the `-l` or `--login` flags

To load environment variables and settings a shell *sources* a number of files.

Sourcing a file is the act of executing it in the current shell. There are 2 ways of sourcing a file:

```
user@darkstar:~$ source filename
```

or using a short notation:

```
user@darkstar:~$ . filename
```

By contrast, executing a file (`./filename`) executes the script in a new shell:

A login shell reads startup files in the following order:

- /etc/profile
- ~/.bash_profile
- ~/.bash_login
- ~/.profile

After executing commands from /etc/profile, the shell checks if the next file exists and is readable. If so, it then executes the commands that the file contains. On a fresh Slackware install, only /etc/profile exists and sets the environment system wide. If a user wants to override their login shell settings, they need to manually create one of the other files. Please note that each of this files is sourced only once when a user logs in.

Please note that the tilde sign (~) in a pathname refers to a user's /home directory. For example, for user fred the ~/ pathname expands to /home/fred/. To access a user's /home directory, the $HOME variable can be used as well. Environment variables will be discussed later.

Interactive Non-login Shell Initialisation

Example: run a terminal window from your desktop or application menu

An interactive non-login shell inherits some environment settings from the login shell startup files and then sources ~/.bashrc if it exists and is readable. By default, the file does not exist in Slackware. A common first step for many Slackware users is to create this file and source /etc/profile to load more environment settings (eg. a more informative command prompt):

~/.bashrc

```
source /etc/profile
```

Please note that, if it exists, ~/.bashrc is loaded each time you start a terminal window.

Non-interactive Non-login Shell Initialisation

Example: run a shell script

A non-interactive non-login shell inherits some variables from the login shell and looks for the environment variable BASH_ENV which points to a startup file to be read by the shell. By default, the BASH_ENV variable is not set in Slackware. If you have some custom environment variables in your ~/.bashrc and you want them read by your scripts as well, you can set this variable in eg. ~/.bash_profile:

```
BASH_ENV=$HOME/.bashrc
```

Configuring Your Environment

While bash provides very reasonable defaults, the power of the command line interface lies in the possibility to customise your working environment to your own preferences and needs. The following

sections will discuss some of the steps you can take to customise your CLI environment.

Do I have to create / edit all of the startup files mentioned above?

No, you don't. See below.

~/.bashrc vs ~/.bash_profile

Admittedly, the fact that `~/.bash_profile` is read by the login shell (ie. only once when you log in) whereas `~/.bashrc` is sourced by non-login interactive shells (each time you start a terminal window) allows for some proper fine tuning of your environment. After all, some variables just need to be assigned once. There is no point in resetting them each time you start a terminal window. For convenience reasons, however, a more common approach is to do keep all your customisations in one file, usually `~/.bashrc`, that will be sourced by both login and non-login shells.

1. Make sure a login shell sources `~/.bashrc` by adding the following to `~/.bash_profile`:

.bash_profile

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

If you are going to do any customisations of root's environment (eg. configuring root's shell prompt discussed below), you need to do the above for root user as well.

2. Place all your customisations in `~/.bashrc`.

That way both interactive non-login and login shells read `~/.bashrc`

~/.bashrc

As it was mentioned above, first we read a global startup file:

```
. /etc/profile
```

You can view all the environment variables that are set with the `env` or `printenv`commands:

```
user@darkstar:~$ env
```

Now we can override some system default settings.

Please note that to implement any changes that you have made in `~/.bashrc`, you need to `source`

it afterwards:

```
source ~/.bashrc
```

Customising Variables

Before we do any customisations, let us mention a few words about `variables`.

`Variables` are containers that hold a value (eg. string or number) that can be later accessed by a shell.

```
user@darkstar:~/bin$ MYVAR="some_value"
user@darkstar:~/bin$ echo $MYVAR
some_value
```

Please note that when we assign some value to a variable we do not use the dollar sign ($) in front of a variable name:

```
VARIABLE="some_value"
```

We use $ when we want to access the value stored in a variable:

```
user@darkstar:~$ echo $VARIABLE
some_value
```

When creating variables it is important to remember the following:

1. variable names can contain alphanumeric characters, as well as the underscore (_). Please note that a variable name cannot start with a number.
2. Do not put any whitespaces around the equal sign (=) or in the variable name.
3. It is recommended to wrap the value of a variable in double quotes (“) to avoid possible problems when referring to its value.
4. Variable names are case sensitive

Exporting a variable

Please consider the following example:

```
MY_VAR="Some text"
export MY_EXPORTED_VAR="More text"
```

By exporting a variable you make it accessible to any sub-shells created in the current shell. In other words, if you want a script to be aware of a variable, you need to export it. If you want to see all the exported variables in the current shell, type:

```
export -p
```

$PATH Variable

The $PATH variable holds a colon (:) separated list of directories containing executable files.

You can use `echo` to check the value of a particular variable. The current value of the $PATH variable on your system should be as follows:

```
user@darkstar:~$ echo $PATH
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/java/bin:/usr/lib/kde4
/libexec:/usr/lib/qt/bin:/usr/share/texmf/bin:.
```

Please note the trailing . (dot). In Bash a dot in a pathname represents the current working directory. For security reasons, it is customary to place the current working directory at the end of your $PATH to avoid overriding system executables.

Users typically store their scripts in `~/bin` so let us create the directory and include it in $PATH.

```
user@darkstar:~$ mkdir ~/bin
user@darkstar:~$ nano ~/.bashrc
```

.bashrc

```
. /etc/profile
PATH=$PATH:~/bin
```

Now you need to `source` the file to put the changes into practice:

```
user@darkstar:~$ source ~/.bashrc
```

$CDPATH Variable

If you work in certain directories on a regular basis, you might want to include them in the $CDPATH variable. Suppose you often work in the `slackbuilds` directory which contains some builds:

```
user@darkstar:~$ cd ~/data/projects/slackbuilds/
user@darkstar:~/data/projects/slackbuilds$ ls
i3 i3status yajl dmenu libev
```

Add it to the $CDPATH variable by modifying `~/.bashrc`:

.bashrc

```
. /etc/profile
PATH=$PATH:~/bin
CDPATH=$CDPATH:~/data/projects/slackbuilds/
```

Now you should be able to cd to any of those directories from any place:

```
user$darkstar:~/.config/xfce4$ pwd
/home/user/.config/xfce4
user@darkstar:~$ cd yajl
/home/user/data/projects/slackbuilds/yajl
```

Set the Default Editor

For historical reasons, there still exist 2 separate environment variables ($VISUAL and $EDITOR) responsible for specifying the default command line editor. You can use any of them or better still, assign the same value to both of them. Bash chooses the default editor by checking the $VISUAL and then $EDITOR. If neither of them is set, it opens Emacs. Add the following to ~/.bashrc (Obviously replace vim with an editor of choice):

```
export VISUAL=vim
export EDITOR=vim
```

Configuring a Shell Prompt

When you first open a terminal window in Slackware, you are likely to see the following shell prompt:

```
bash-4.2$
```

It is quite bare and the only thing it tells us is that we run Bash version 4.2 and that we run it as a standard user (as opposed to root). The moment we source the system-wide configuration file, we will get a considerably more informative shell prompt displaying the current user, the hostname as well as the current working directory:

```
username@hostname:~$
```

By default the primary Slackware bash prompt is configured in /etc/profile and stored in the PS1 variable:

```
PS1='\u@\h:\w\$ '
...
PS2='> '
```

You may also notice the PS2 variable which stores a secondary shell prompt (>) which is used when we split a very long command with the backslash (\) or when the shell is waiting for further input before interpreting the command, as in:

```
user@darkstar:~$ ./configure --prefix=/usr --sysconfdir=/etc --enable-
warnings \
> --disable-atsui --disable-gtk-doc --disable-glitz --disable-quartz --
disable-static \
> --disable-win32 --disable-xcb
```

```
user@darkstar:~$ echo "This is a piece
> of text. The moment I close the string
> with a double quote, bash will execute the command."
This is a piece
of text. The moment I close the string
with a double quote, bash will execute the command.
user@darkstar:~$
```

Please note that there also exist variables PS3 and PS4 which store the `select` command and execution trace prompts, respectively (more information: `man bash`).

Display the Current Prompt Settings

To display the current prompt settings, you can echo the contents of the PS1 variable:

```
user@darkstar:~$ echo $PS1
\u@\h: \w\$
user@darkstar:~$ echo $PS2
>
```

As you can guess from the example above, `\u` stands for the username, `\h` for the hostname, and `\w` represents the current working directory pathname. The trailing `\$` expands to # if you are `root` (ie. the effective UID is 0) and $ in all other cases. Have a a look at this table of some common special characters used in the PS1 variable:

Character	Description
\u	username of the current user
\h	hostname (short output - to the first .)
\H	FQDN (eg. darkstar.domain.com)
\w	full pathname of the current working directory
\W	the basename of the current working directory
\d	current date
\t	current time in 24-hour (HH:MM:SS) format
\T	current time in 12-hour format
\!	current command history number
\j	number of jobs that the current shell manages

Setting the Prompt

You can test some prompt configurations by temporarily defining the variable directly in the shell

```
user@darkstar:~$ PS1="\u@\h \! \j \t \$ "
user@darkstar 503 0 20:30:24 $
```

Apart from the standard user and host names, the prompt now displays the bash history number, the number of jobs running in the background, as well as the current time. You can get creative when designing your own shell prompt, especially with the help of the command substitution $(…) construct:

```
user@darkstar:~$ PS1="\u@\h [\$(ls | wc -l)]:\$ "
user@darkstar[9]:$
```

The above shell prompt will also display the number of files and directories in the current working directory. It is important, however, to remember that the command prompt is there to help you. For that reason, one needs to keep the balance between keeping the prompt informative and practical at the same time. A very descriptive, but long, command prompt may easily hinder your productiveness.

Prompt in Colours

One way to display the shell prompt in colour is to use the `tput` command which checks the terminfo database and generates relevant codes for a terminal. It has a number of capabilities including:

- `setaf` - set ANSI foreground
- `setab` - set ANSI background

Its syntax is as follows:

Command	Description
tput setaf colour_code	Set a foreground colour
tput setab colour_code	Set a background colour
tput sgr0	Switch off any colours

Colour Codes

The values of colour codes are as follows:

Colour	Colour Code
black	0
red	1
green	2
yellow	3
blue	4
magenta	5
cyan	6
white	7

Configure the Prompt in Colour

Let us define some colour variables in `~/.bashrc`:

```
# Defining foreground variables
P_BLACK="\[$(tput setaf 0)\]"
P_RED="\[$(tput setaf 1)\]"
P_GREEN="\[$(tput setaf 2)\]"
P_YELLOW="\[$(tput setaf 3)\]"
P_BLUE="\[$(tput setaf 4)\]"
```

```
P_MAGENTA="\[$(tput setaf 5)\]"
P_CYAN="\[$(tput setaf 6)\]"
P_WHITE="\[$(tput setaf 7)\]"
P_RESET="\[$(tput sgr0)\]"
```

Please note that we wrap `\[…\]` around the output of `tput` to indicate that it contains non-printable characters. This helps avoid any problems with calculating line wraps.

Having defined the colour variables, we can go on to configure the prompt:

```
PS1="$P_BLUE\u@\h: \w \\$ $P_RESET"
```

This will produce a blue prompt. Please note that we need to turn off any colours at the end to prevent the commands we type from being blue as well.

The next prompt gives each section of the prompt a different colour:

```
PS1="$P_GREEN\u$P_RED@$P_YELLOW\h:$P_CYAN\w$P_BLUE\\$ $P_RESET"
```

Bear in mind that your output might look differently depending on a terminal and its colour settings in eg. `~/.Xdefaults`.

Root Prompt

It is useful to give the `root` user a different prompt (the # character will be red) by adding the following (+ colour definitions) to the root's `.bashrc`:

```
PS1="$P_GREEN\u$P_RED@$P_YELLOW\h:$P_CYAN\w$P_RED\\$ $P_RESET"
```

Do not forget to source root's `.bashrc` in its `.bash_profile` to make sure it works for both non-login (`su`) and login (`su -`) shells.

Prompt Background Colour

If you'd like to set the prompt's background colour, you can also add the `setab` capability. Here's an example (without the use of variables):

```
PS1="\[$(tput setab 4)$(tput setaf 7)\]\u@\h:\w $ \[$(tput sgr0)\]"
```

Permanently Storing the Prompt

Once you are happy with your bash prompt, you can permanently store it in `~/.bashrc`:

```
export PS1="\u@\h \! \w\$ "
```

Building Aliases

Aliases are shortcuts or abbreviated commands used in a shell in order to avoid typing long commands. Aliases are usually created to modify existing commands by adding some flags or to join a few commands in order to create new custom commands.

An example of an alias would be:

```
alias ll='ls -l'
```

Now typing `ll` calls the `ls` command with the `-l` (or `long listing format`) flag.

Another example would be creating a shortcut to query installed packages on a Slackware system:

```
alias qp='ls /var/log/packages | grep'
```

```
user@darkstar:~$ qp emacs
emacs-24.2-x86_64-1
```

The general syntax of an alias is:

```
alias name='full command'
```

To create a temporary alias for the current session, just define it directly on a command line. It will not be remembered when you start another session. To store an alias permanently to be accessible by the shell in the future, place it in `~/.bashrc`.

If you need to (temporarily) switch off an alias you can use the `unalias` built-in:

```
user@darkstar:~$ qp emacs
emacs-24.2-x86_64-1
user@darkstar:~$ unalias qp
user@darkstar:~$ qp
bash: qp: command not found
```

If you run a (long) command on a regular basis, it might be convenient to create an alias for it.

Have a look at some more examples of aliases below. This may help you create your own aliases.

Please remember that creating aliases that replace system commands may introduce security risks so always consider creating a new command instead.

```
alias e='exit'
alias ll
alias lla='ls -al | less'
alias mkdir='mkdir -pv'
alias rm='rm -i'
```

```
alias 1.='cd .. ; pwd'
alias 2.='cd ../.. ; pwd'
alias 3.='cd ../../.. ; pwd'
alias 4.='cd ../../../.. ; pwd'
alias h='history'
alias eq='alsamixer -D equal'

alias emc='emacs -nw'
alias org='emacs -nw /home/user/data/todo.org'
alias ci3='vim /home/user/.i3/config'

alias psp='ps aux | grep'
alias wic='nmap -sP 192.168.1.0/24'

alias dl='cd /home/user/data/downloads/ ; ls'
alias mftp='lftp sftp://user@ftp.server.com'
alias mylaptop='ssh user@ip_address'
alias myserver='ssh user@ip_address'

alias t='task'
alias r2j='mkdir jpg; ufraw-batch --out-type=jpeg --out-path=./jpg ./*.NEF'
```

Aliases and Security

Sometimes aliases may pose a security risk in a sense that they can spoof other commands (eg. on compromised systems). Consider the following:

```
alias ls='some_nasty_command'
```

To view currently set aliases, just type `alias` at the command prompt. To clear all the aliases, type:

```
\unalias -a
```

Escaping the command with \ prevents any alias expansion in case someone had tried to spoof the `unalias` command. See the following:

```
user@darkstar:~$ pwd
/home/user
user@darkstar:~$ alias pwd='echo 666'
user@darkstar:~$ pwd
666
user@darkstar:~$ \pwd
/home/user
```

Creating functions

Fix Me!

~/.bashrc

At the moment our ~/.bashrc looks as follows. At the bottom you'll see some additional settings that have not been discussed. The comments above them should clarify their meaning.

```
# load the system-wide environment
source /etc/profile

# Add a directory with your scripts to the path.
PATH=$PATH:~/bin

# Configure the CDPATH variable to include a frequently visited directories
CDPATH=$CDPATH:~/data/projects/slackbuilds/

# Set the default editor
export VISUAL=vim
export EDITOR=vim

# Defining foreground variables for the prompt
P_BLACK="\[$(tput setaf 0)\]"
P_RED="\[$(tput setaf 1)\]"
P_GREEN="\[$(tput setaf 2)\]"
P_YELLOW="\[$(tput setaf 3)\]"
P_BLUE="\[$(tput setaf 4)\]"
P_MAGENTA="\[$(tput setaf 5)\]"
P_CYAN="\[$(tput setaf 6)\]"
P_WHITE="\[$(tput setaf 7)\]"
P_RESET="\[$(tput sgr0)\]"

# Setting a fancy prompt for the current user
export PS1="$P_GREEN\u$P_RED@$P_YELLOW\h:$P_CYAN\w$P_BLUE\\$ $P_RESET"

# Setting aliases
alias e='exit'
alias ll='ls -l'
alias lla='ls -al | less'
alias mkdir='mkdir -pv'
alias rm='rm -i'
alias 1.='cd .. ; pwd'
alias 2.='cd ../.. ; pwd'
alias 3.='cd ../../.. ; pwd'
alias 4.='cd ../../../.. ; pwd'
alias h='history'
alias eq='alsamixer -D equal'

alias emc='emacs -nw'
alias org='emacs -nw /home/user/data/todo.org'
alias ci3='vim /home/user/.i3/config'

alias psp='ps aux | grep'
```

```
alias wic='nmap -sP 192.168.1.0/24'

alias dl='cd /home/user/data/downloads/ ; ls'
alias mftp='lftp sftp://user@ftp.server.com'
alias mylaptop='ssh user@ip_address'
alias myserver='ssh user@ip_address'

alias t='task'
alias r2j='mkdir jpg; ufraw-batch --out-type=jpeg --out-path=./jpg ./*.NEF'

#########################################################################
#                       Additional settings:                            #
#########################################################################

# Specify an NNTP Server
export NNTPSERVER='aioe.org'

# To take advantage of multicore CPUs you can use the MAKEFLAGS variable.
# For example the equivalent of "make -j2" would be:
# export MAKEFLAGS="-j2"
# Uncomment the above line to use it.
```

Other configuration files

~/.bash_logout

~/.inputrc

~/.netrc

Wildcards

Shell history

Useful Keybindings

Sources

- Originally written by Marcin Herda

work in progress, author sycamorex
[1)]

When a shell is invoked with the `-l` or `—login` flags, it acts as a login shell