Bash Configuration

Shell Dressing

For many newbies the shell is a mystery best left to advanced users and Linux gurus. The shell prompt provides access to a flexible working environment that allows users to perform almost any task. BY ANDREAS KNEIB

The history of the UNIX shell starts at the end of the 60's. At this time Ken Thompson and Dennis Ritchie were still busy developing the UNIX operating system. The developers were still on the lookout for an interface between the system and its users.

This interface would need to pass information between the operating system and the user. The machine should be able to understand commands entered on the keyboard as easily as the user could understand messages generated by the system. This is why the shell is referred to as a command interpreter.

Enter Stephen R. Bourne and his invention, the Bourne Shell (aka the AT&T Shell or simply *sh*), which has become the classic UNIX shell since. The shell provided scripting facilities, although its interactive functionality was somewhat impractical. It did not provide or even envisage a history function – that is, a function that allows users to see and edit commands they have already used. More developments followed, such as the Korn Shell (*ksh*) and the C Shell (*csh*).

In the early 80's, Richard Stallman's GNU project was faced with a similar problem to the one Thompson and Ritchie had been faced with: he needed a user interface.

His answer was to write a command interpreter of his own, which he called Bash. This name is both an acronym and a play on words and means Bourne Again Shell. Bash command structure is based on the Bourne Shell, but also incorporates some elements from the Korn and C shells.

Besides interpreting commands, bash supports the user by calling other programs. Its powerful programming language also allows the user to script complex tasks.

If you use a major distribution, you will probably find a whole bunch of shell scripts running in the background. These files are identified by the *#!/bin/bash* or *#!/bin/sh* string in the first line. The expression specifies the path to the command interpreter and is known to programmers as shebang (which is short for sharp, "#" and bang, "!").

Keys and Modes

Commands entered in the command line can be edited before pressing enter to launch them. There are two text editing modes: vi and emacs. Both UNIX standard editors use a variety of keyboard shortcuts to delete, enter or search for data, and these shortcuts are also available in bash.

You can toggle the GNU shell, depending on your preferred editor: *set -o vi* will switch to vi mode, and *set -o emacs* will return to the default emacs

Table 1: Emacs keyboard mappings in bash

Keys	Command
[Ctrl-r]	Search for commands
[Ctrl-l]	Clear screen
[Tab]	Complete command or filename
[Ctrl-e]	Place cursor at end of line
[Ctrl-a]	Place cursor at beginning of line
[Ctrl-u]	Delete input
[Ctrl-k]	Delete input as of cursor position
[Ctrl-d]	Delete character under cursor
[Alt-t]	Transpose words around cursor
[Alt-b]	Move cursor back one word
[Alt-f]	Move cursor forward one word

mode. Figure 1 shows a user switching editor modes. *bind -v | grep keymap* tells you the current mode. To keep things simple, we will only be looking at emacs mode in this article.

Let's take a look at some examples of keyboard shortcuts. You can hit [Alt-t] to swap the two words before and after the cursor position. The up and down arrow keys allow you to scroll through the commands you have just entered. The [Tab] key expands commands and/or filenames, and [Ctrl-r] searches for earlier entries. Table 1 provides an overview of the most important keyboard shortcuts.

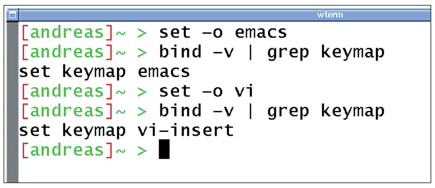
Global keyboard mappings are stored in the */etc/inputrc* file, or in *~/.inputrc* for individual users. Of course, these configuration files do not merely contain the bash configuration. Instead they control a function called *readline* that is used by other programs (such as GNU Plot, the GNU Debugger *gdb*, or even sax2 and smbclient). Let's take a look at the internal structure of the */etc/inputrc* file. You can expect entries like the following:

```
# /etc/inputrc examplefile

set meta-flag on
set output-meta on
set convert-meta off
"\e[A":          previous-history
"\e[4~":         end-of-line
```

The first three lines starting with *set* do exactly that with readline variables. The *set meta-flag on* entry allows you to input 8 bit characters (that is foreign language characters and the like). *set output-meta on* allows the shell to output these characters.

The last variable, *convert-meta off*, prevents unwanted character conversions. As you may already have guessed, this variable allows you to use non-



```
[andreas]~ > set -o emacs
[andreas]~ > bind -v | grep keymap
set keymap emacs
[andreas]~ > set -o vi
[andreas]~ > bind -v | grep keymap
set keymap vi-insert
[andreas]~ > █
```

Figure 1: Switching editor modes in bash

standard characters in the command line.

The lines that follow are more difficult to interpret. The *previous-history* scrolls back through the list of previously entered commands. "\e[A" maps the up-arrow key to this parameter, with *\e* being the Escape character. A tool such as *showkey* might help you get to grips with this configuration: it displays the following when you press the up-arrow and end-of-line keys:

```
[and]~ > showkey -a
^[[A       27  0033  0x1b
           91  0133  0x5b
           65  0101  0x41
^[[4~      27  0033  0x1b
           91  0133  0x5b
           52  0064  0x34
          126  0176  0x7e
```

^[[4~ maps to the *"\e[4~"* definition in */etc/inputrc*. It uses the *end-of-line* function to specify moving the cursor to the end of the current line. *^[[4~* or *"\e[4~"* maps to the [End] key. Our example may be slightly different than your configuration, as the X Server configuration is also significant. There are a few GUI programs, such as *xev* and *xkeycaps* that you might like to use as an alternative to *showkey*.

Let us stop editing the configuration file at this point and try out a few additional settings with the *bind* command. If you need more detail on the readline library, you can of course type *man readline* to display the manual.

Binding and Unbinding

The *bind* command provides a convenient keyboard mapping method for the shell. You can use the *-P* option to display a list of the current bindings as in Listing 1.

Our example uses | (the 'pipe' character) to redirect the output from *bind -P* to the *less* pager for more convenient scrolling and reading. The technical term for this is piping.

As the previous example shows, the [Ctrl-b] or left arrow keys call the *backward-char* function. \C denotes the Ctrl key.

The *bind key function* command allows you to re-map keys during a session. As you may already have noticed, the [Ctrl-l] keyboard shortcut clears the screen and restores the prompt.

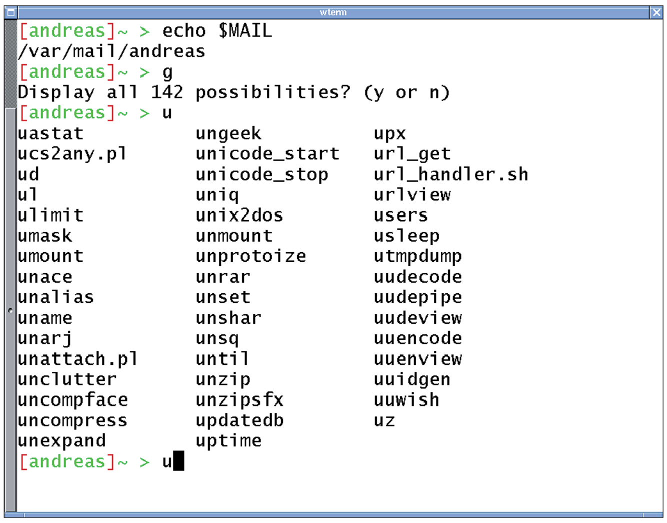
Now let's bind the refresh function to a new keyboard shortcut. We will start by piping the output of the bind command to *grep* to filter the line with the *C-l* character sequence:

```
[and]~ > bind -p | grep "C-l"
"\C-l": clear-screen
```

As we can see the keyboard shortcut [Ctrl-l] maps to the *clear-screen* function. Just to make sure, let's now call the function with the bind option, *-q*. The parameter *-q* stands for query and shows

Listing 1: Current bindings

```
[and]~ > bind -P | less
abort can be found on "\C-g", "\C-x\C-g", "\e\C-g".
accept-line can be found on "\C-j", "\C-m".
backward-char can be found on "\C-b", "\e[D".
backward-delete-char can be found on "\C-h", "\C-?".
[...]
```



```
[andreas]~ > echo $MAIL
/var/mail/andreas
[andreas]~ > g
Display all 142 possibilities? (y or n)
[andreas]~ > u
uastat          ungeek          upx
ucs2any.pl      unicode_start   url_get
ud              unicode_stop    url_handler.sh
ul              uniq            urlview
ulimit          unix2dos        users
umask           unmount         usleep
umount          unprotoize      utmpdump
unace           unrar           uudecode
unalias         unset           uudepipe
uname           unshar          uudeview
unarj           unsq            uuencode
unattach.pl     until           uuenview
unclutter       unzip           uuidgen
uncompface      unzipsfx        uuwish
uncompress      updatedb        uz
unexpand        uptime
[andreas]~ > u█
```

Figure 2: Bash command completion using the tabulator key

the keyboard shortcut that is bound to the specified function.

```
[and]~ > bind -q clear-screen
clear-screen can be invoked ↩
via "\C-l".
```

The *-u* (unbind) parameter removes the bindings for *clear-screen*. The following syntax is designed to test whether the modification has been applied:

```
[and]~ > bind -u clear-screen
[and]~ > bind -q clear-screen
clear-screen is not bound to ↩
any keys.
```

The next step is to apply the *bind key function* syntax and ensure that [Ctrl-t] really does clear the screen:

```
[and]~ > bind '"\C-t" ↩
clear-screen'
[and]~ > bind -q clear-screen
clear-screen can be invoked ↩
via "\C-t".
```

Thus, [Ctrl-t] now clears the bash screen, rather than [Ctrl-l]. Unfortunately, this overwrites the *transpose-chars* function. Obviously, it makes sense to avoid overwriting existing bindings, something you should remember for the future.

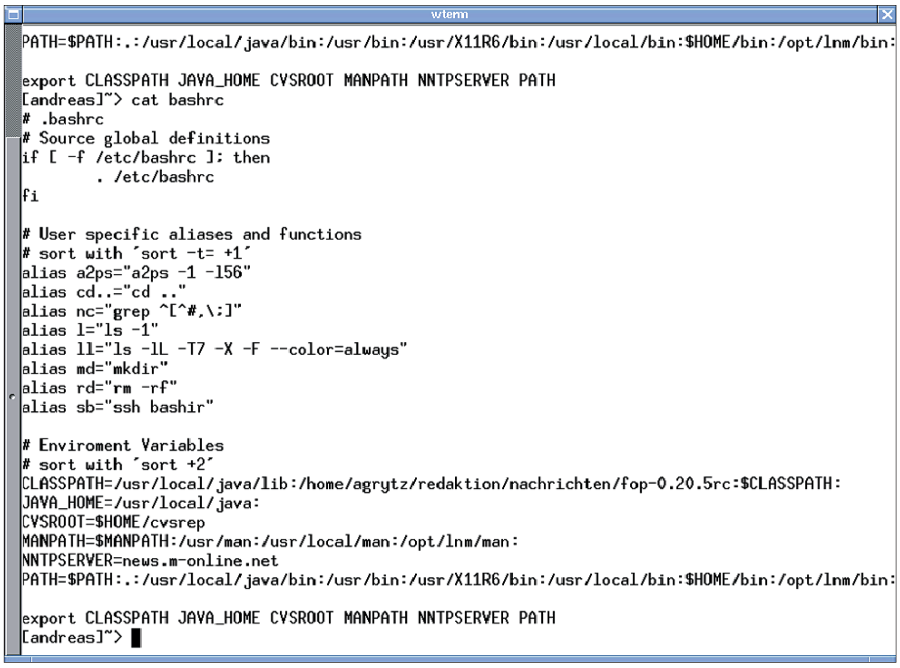
You can refer to Table 2 for a list of bindings or consult the comprehensive manual by typing *man bashbuiltins*.

Files and More Files

Now that we have introduced the */etc/inputrc* options, let's take a look at some more bash files. The GNU shell uses a variety of configuration files, and all of them are used differently. SuSE,

Table 2: Bind Options

Option	Action
bind -f [file]	Reads [file] as the configuration file
bind -l	Function list
bind -p	List of functions and bindings
bind -P	Simple list of functions and bindings
bind -q	Queries the keyboard combination for a function
bind -r key	Removes the bindings for a key
bind -u function	Removes the keyboard shortcut for a function
bind -V	Shows the readline variables



```
PATH=$PATH:.:/usr/local/java/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:$HOME/bin:/opt/lnm/bin:

export CLASSPATH JAVA_HOME CVSROOT MANPATH NNTPSERVER PATH
[andreas]~> cat bashrc
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
# sort with 'sort -t= +1'
alias a2ps="a2ps -1 -156"
alias cd..="cd .."
alias nc="grep ^[^#,\:]"
alias l="ls -1"
alias ll="ls -lL -T7 -X -F --color=always"
alias md="mkdir"
alias rd="rm -rf"
alias sb="ssh bashir"

# Enviroment Variables
# sort with 'sort +2'
CLASSPATH=/usr/local/java/lib:/home/agrytz/redaktion/nachrichten/fop-0.20.5rc:$CLASSPATH:
JAVA_HOME=/usr/local/java:
CVSROOT=$HOME/cvsrep
MANPATH=$MANPATH:/usr/man:/usr/local/man:/opt/lnm/man:
NNTPSERVER=news.m-online.net
PATH=$PATH:.:/usr/local/java/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:$HOME/bin:/opt/lnm/bin:

export CLASSPATH JAVA_HOME CVSROOT MANPATH NNTPSERVER PATH
[andreas]~> ▌
```

Figure 3: The .bashrc file is used for shell configuration

Red Hat, and other distributions often do things their own way.

So let's try to keep to what the bash manual *man bash* – a mere 5100 lines – tells us. When bash is launched as a login shell it parses the following configuration files in the following order:

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

The question is, what makes bash a login shell? After logging on via a character based console, you will discover that you have access to a shell. Bash has parsed the commands in the four files we just mentioned and launched them when you logged on.

It makes sense to place entries such as environment variables in the global */etc/profile* or the local *~/.profile*, although the shell will not parse the later unless *~/.bash_profile* is missing. Any subsequent shells inherit their settings and thus the environment from bash.

Things are a bit more confusing if you use a display manager instead of a text console to log on. Your window manager preferences define whether an *xterm* with bash is launched as your login shell; the window manager itself is a program that launches X Window applications.

Calling an Xterm with the *-ls* flag set is the best way to ensure access to a login shell. Any configuration changes you make during a session take effect after calling *source* to re-parse the configuration file. The syntax is as follows:

```
[root] # source /etc/profile
```

However, you cannot do this on an X Window System if the shell is running in a terminal emulation. The new configuration does not affect the shell and it will continue to use the original settings. Your best bet is to quit the GUI desktop and log back on again.

Automated tasks, such as removing old backup files, that you want bash to perform when you log out should also be placed in *~/.bash_logout*. That concludes our tour of the login shell, although we have hardly touched on its many capabilities, as they are beyond the scope of this article.

As you may have guessed there are also non-login shells. Bash will expect to find instructions for this kind of shell in *~/.bashrc*. You should not rely on this, as many distributions also parse the file for login shells.

The login shell configuration may be so good that you do not need to modify the bash configuration in *~/.bashrc*. Now its time to look at the Babylonian confusion of distributions using SuSE as an example. The */etc/profile* header contains a note from the distributor, telling you not to edit this file. Any global changes you want to make must be placed in */etc/profile.local*, the reason being that */etc/profile* can be overwritten by a system update or the SuSE YaST tool.

That's not all: */etc/profile.dos*, */etc/SuSEconfig/profile*, and any files in the */etc/profile.d* directory are parsed by */etc/profile*.

Inexperienced users are advised to steer clear of these files. As you gain more experience, you should be able to break down and simplify these structures.

Variables and Environments

We have looked at configuration so far, but now it is time to examine their contents more closely. *alias* is a function that many users utilize; it allows you to define a command as a shortcut for a more complex command line:

```
[and]~ > alias gnus='xemacs -nw↩
 -f gnus'
```

This syntax creates the *gnus* shortcut that launches the Gnus newsreader in a character based console without too much typing. Of course, you can assign an alias to a series of semicolon separated commands:

```
[and]~ > alias unmount=↩
'cd;umount /media/cdrom;eject'
```

Typing *unmount* in the command line now changes (*cd*) to your home directory, unmounts the medium in your CD ROM drive (*umount /media/cdrom*) and ejects the CD (*eject*).

However, you cannot use an *alias* to pass a parameter to a program:

```
[and]~ > alias option=↩
'-fn lucidasanstypewriter-14'
[and]~ > xterm option
xterm:  bad command line ↩
option "option"
```

As this attempted alias binding does not work, it makes sense to remove the shortcut:

```
[and]~ > unalias option
```

Typing *alias* without any additional parameters displays a list of aliases in alphabetical order. As *alias option* = … is no longer on the list, it is safe to assume that we have removed the binding:

```
[and]~ > alias
alias newart='vim ~/work/↩
article/newart.txt'
alias nhol='sudo /usr/local/↩
sbin/fetchnews -vvv'
alias o='less'
alias rd='rmdir'
```

Any aliases you want to make persistent should be added to *~/.bashrc* or *~/.profile*.

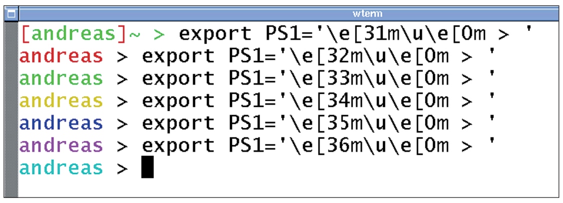
We mentioned environment variables just a while back. These variables are passed to commands and programs and provide an environment containing specific constants such as the standard news server. Let's look at an example to discover what this implies.

Mutt is a character based mail program that requires an external editor for message editing. The mailer looks at the *EDITOR* variable to discover the standard editor. To find out what your editor variable defaults to, simply type the following command:

```
[and]~ > echo $EDITOR
/usr/bin/vim
```

PATH and *MANPATH* are two other important environment variables. *PATH* stores the search path for executable files. When you launch a program from bash, the shell searches for the file in a list of directories. The search terminates when the required entry is found, and the command is executed. Thus, the position of a directory in the *PATH* is significant.

MANPATH on the other hand stores the search path for manpages. If this has whetted your appetite, you can type *export* and *printenv* to view an almost infinite list of environment variables.



```
[andreas]~ > export PS1='\e[31m\u\e[0m > '
andreas > export PS1='\e[32m\u\e[0m > '
andreas > export PS1='\e[33m\u\e[0m > '
andreas > export PS1='\e[34m\u\e[0m > '
andreas > export PS1='\e[35m\u\e[0m > '
andreas > export PS1='\e[36m\u\e[0m > '
andreas > █
```

Figure 4: Prompting in color

It is not difficult to create your own environment variables – the process typically involves the *export* command. Let's start by defining a variable called *PS1*. It will contain a prompt string. The prompt is the part of the command line that precedes the area where commands are entered. The prompt can provide you with useful information:

```
[and]~ > export PS1='\u at \t ↩
on \h in \w \$ '
and at 09:29:02 on comone in ~ $
```

As you can see, the prompt has changed. The *\u* displays the user name, *\t* shows the 24 hour time-of-day, *\h* tells you the hostname, *\w* outputs the current directory, and *\$* terminates the prompt. If this is too untidy for you, you can simply display the user name, using a background color to highlight it. To do so, we will be using the escape character *\e* and some escape sequences:

```
[and]~ > export PS1=↩
'\e[32m\u\e[0m \$ '
```

This results in a green prompt that helps you keep track of what bash is doing when the screen becomes cluttered. For more information on Escape sequences, simply type *man console_code*.

You might like to add your favorite prompt to */etc/profile* or *~/.bash_profile* to make it persistent:

```
PS1='\w \$'
export PS1
```

Bash is neither a mystery nor a hacker tool, but simply a flexible program that helps you get to grips with every day computing. ■

THE AUTHOR

Andreas Kneib has been living on this planet since 1967, and if he's not too busy with his vegetable patch, tends to sit in the corner messing with scripts and configuration files. His first contact with computers was an old C64, followed by an Amiga 500, and Linux some time later. The author has been writing for Linux Magazine on and off since 2001.